Budapest University of Technology and Economics
Department of Networked Systems and Services

Improved security and protection from malware for embedded IoT devices

Collection of Ph.D. Theses
of
Dorottya Futóné Papp

Supervisor: Levente Buttyán, habil. Ph.D.

Budapest, Hungary
2020

Acknowledgement

The research presented here received the financial support from the following projects, programmes, and funding agencies:

- SETIT Project (2018-1.2.1-NKP-2018-00004)[1]
- The H2020-ECSEL programme under grant agreement no. 692474 (AMASS).
- SECREDAS project, which received funding from ECSEL Joint Undertaking as part of the European Union's Horizon2020 Research and Innovation Programme
- EU ARTEMIS project EMC^2, funded by the Austrian Research Promotion Agency (FFG), as well as the H2020-ECSEL programme

I also gratefully acknowledge the funding from the EFOP grant (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications), co-financed by the European Social Fund.

1 Introduction

Embedded devices are special-purpose devices developed to perform specific tasks. They are present in many modern-day application domains, including healthcare, transportation and agriculture. A recent advancement in the field of developing embedded devices is their connection to the Internet, which leads to what is known as the Internet of Things (IoT for short). Internet connection has enabled a wide range of new and innovative applications for these devices, which we commonly call embedded IoT devices. For example, houses equipped with smart meters can automatically report water and energy use. Smart traffic lights in cities can sense the flow of traffic and adjust accordingly to reduce traffic jams. Medical experts can monitor certain implanted healthcare devices, e.g. pacemakers, remotely.

Internet connection, however, has also opened the way for attackers to target and compromise embedded IoT devices. Attacking these devices is a rational choice from the attackers' point of view. Embedded IoT devices, especially in the healthcare and smart home application domains, handle sensitive and personal data worth stealing. In addition, even though the computational power of individual embedded IoT devices is small, it is non-negligible when considering these devices combined. From the technical point of view, the hardware and software components of embedded IoT devices are not very different from those found in traditional computers. Their insecure configuration, e.g., accessible open ports without proper authentication, as well as default or hard-coded passwords, allow easy access the device. It is also technically possible to exploit vulnerabilities in software components running on IoT devices, including their firmware

[1]Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

and operating system (OS), which is often based on some embedded Linux variant. Consequently, the security community has observed a rise in the number of viruses, worms, Trojans and other types of malware targeting embedded IoT devices. One of the most infamous examples is Mirai [2], which infected hundreds of thousands of IoT devices and launched one of the largest distributed denial of service attacks ever recorded against popular Internet-based services in 2016. The IoT threat landscape, however, includes other malware as well, for example, Gafgyt, Tsunami, and Dnsamp [10].

In my dissertation, I explore the security of embedded IoT devices. Specifically, I review the threat landscape of IoT devices with a new taxonomy for attack scenarios and highlight several areas in which improvements to the security of embedded devices are necessary. One of the key insights gained from applying my proposed taxonomy to existing vulnerability data is that malware is a major threat to embedded IoT devices. Therefore, I focus my attention on this threat and study the issue of malware from two aspects. First, I address the problem of malware clustering. Malware clustering allows for focusing malware analysis on only those samples that have not been analyzed before and are not similar to any known malware. Second, I address the problem of stealthy malware. There exists a class of malware that perform malicious actions only when specific inputs, e.g., commands from the attacker, are received. This behavior is known as trigger-based behavior and is a great challenge for analysts. In order to overcome this challenge, I propose a new analysis method to uncover environmental constraints guarding malicious behavior. Finally, I propose a new mode of operation, RoViM, which allows embedded IoT devices to restore themselves periodically to a known compromise-free state, while keeping their functionality/services continuously available.

2 New Results

This section is a short summary of the results that are discussed in details in my dissertation.

2.1 The threat landscape of embedded IoT devices

Having a comprehensive view and understanding of an attacker's capability, i.e., knowing the enemy, is a prerequisite for security engineering embedded IoT systems. Security analysis, secure design, and secure development must all take into account the full spectrum of the threat landscape in order to identify security requirements, as well as innovate and apply security controls within the boundary of constraints. Understanding the threat landscape requires identifying the main causes of successful attacks, the attacks' commonalities, and the main vulnerabilities.

THESIS 1: *I reviewed the threat landscape of embedded IoT devices and proposed a new attack taxonomy to systematically identify and classify common attacks in [C1]. I evaluated the proposed attack taxonomy on relevant records from the CVE database in [C2][2] and highlighted several important aspects of the security of embedded devices.*

Due to researchers' specific interests, the cost incurred in security testings, and the non-disclosure agreement forced by the vendors or asset owners, the published attacks/hacks only reflect a fraction of the whole threat landscape. To gain a comprehensive view, we must study the information on vulnerabilities related to embedded systems. The main information source is the Common Vulnerabilities and Exposures (CVE) database[3], the most comprehensive aggregation of security vulnerabilities. Each entry in the CVE database is assigned a standardized identifier, which can be used to share vulnerability information across different organizations. At the time of my research (2015), the database contained more than 60,000 entries, not all related to embedded systems. I improvised several techniques on-the-fly-to filter and extract relevant entries from the general vulnerability data, and manually analyzed the selected entries. The result of the analysis was a set of attack classification criteria that served as a basis for the attack taxonomy.

Based on my analysis of the CVE records and prior work on attack taxonomies, I defined 5 dimensions along which attacks against embedded systems can be classified: (1) precondition, (2) vulnerability, (3) target, (4) attack method, and (5) effect of the attack. The precondition dimension contains possible conditions that are needed to be satisfied in order for the attacker to be able to carry out the attack. The vulnerability dimension contains different types of vulnerabilities that can be exploited by the attacker. The target dimension contains possible attack targets by which I mean a specific layer of the system architecture or the embedded device as such if no specific layer can be identified as a target. The attack method dimension contains various exploitation techniques that the attacker can use. The effect dimension contains possible impacts of an attack.

I evaluated the proposed taxonomy on 3,826 CVE entries related to embedded systems. Figure 1 shows the results on a parallel coordinates diagram. Each path on the diagram is an attack scenario derived from individual CVE results, the categories the path touches in each dimension show a different aspect of the attack. The thicker a path is, the more CVE entries mention it as a possible scenario. Given the recent trends in machine-to-machine communication and the growing number of embedded IoT devices, I expect Internet facing devices to continue to suffer the majority of attacks. These attacks can come in various forms; however, based on Figure 1, malware and exploits are the most common approaches which have to be dealt with.

[2] In the case of both papers, Zhendong Ma supervised my work for AIT Austrian Institute of Technology GmbH.

[3] https://cve.mitre.org/ (Last visited: Jan 8, 2020)

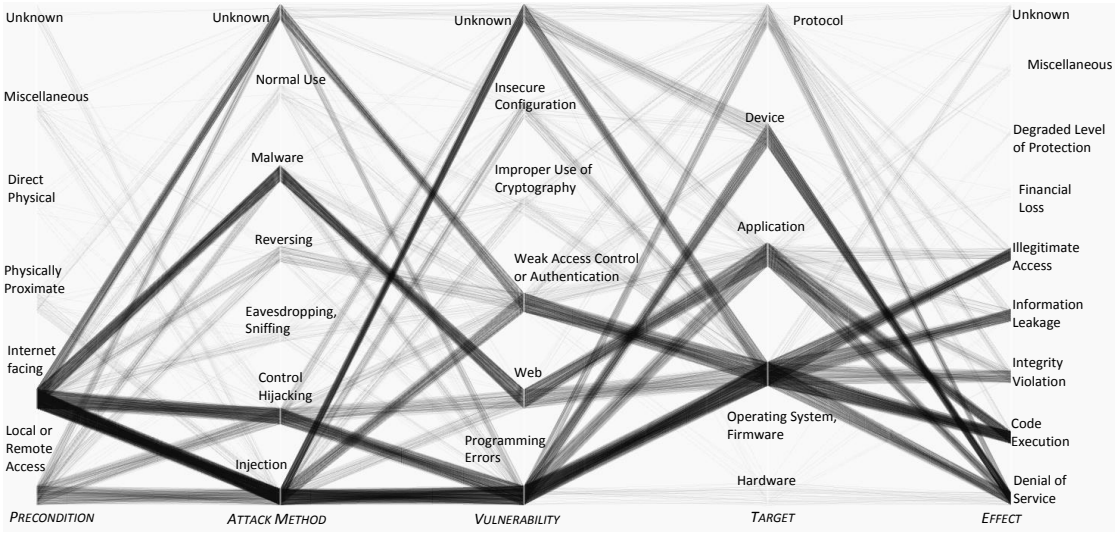

Figure 1: Common attack scenarios for embedded IoT devices

2.2 Malware clustering using binary similarity hashes

Based on results presented in Section 2.1, the security of embedded IoT devices could be improved significantly with better protection from malware, such as Mirai [2]. Anti-virus companies rely on malware classification methods [27, 26, 12] to identify relating malware samples. Clustering malware into families makes sense, as members of the same family, while being different at the binary level, exhibit similar behavior. Ultimately, such clustering reduces the load on analysts by allowing them to focus on samples that are not similar to any known sample.

THESIS 2.1: *I studied the possibility of clustering malware samples based on their binary similarity hashes, specifically, their TLSH [23] similarity score in [C3][4]. I showed that existing clustering algorithms, k-medoid and OPTICS, have unacceptable performance on a large corpus of IoT malware.*

k-medoid [15] is a PAM-based algorithm in which clusters can have only valid data points as their centers (also called medoids). The algorithm has one input parameter, k, which determines how many clusters will be present in the output of the algorithm. The algorithm first selects k medoids, then tries to fit all data points to the nearest cluster head. Medoid selection and re-clustering is repeated iteratively until an optimum is reached. The measure of goodness for the algorithm is $s(k)$, which measures the gain in assigning data points to specific clusters based on distance. The closer the metric is to 1, the better the setup.

[4] Márton Bak implemented to data collection methodology and clustering algorithm. Csongor Tamás provided the empirical TLSH threshold for detecting variants of malware families.

Table 1: Best $s(k)$ values for different k-medoid cluster configurations

k	$s(k)$
318	0.962
625	0.959
232	0.955
925	0.950
396	0.948

Table 2: Statistics of k-medoid cluster configuration with $k = 17$

s(k)	0.405
Maximum diameter	1,038
Minimum diameter	180
Mean diameter	467.824
Largest cluster size	1,110
Smallest cluster size	35
Mean cluster size	573.294

There are several rationales behind choosing k-medoid as the clustering algorithm. It is unsupervised, ie. there is no need to supply any additional data, only the similarity measurements between samples. In addition, the algorithm only selects existing data points as cluster heads. This is useful for malware clustering, because cluster heads can represent other malware samples in the same cluster. The disadvantage of this algorithm is that the input k has to be specified manually.

Unfortunately, I do not know how many variants there are in the data set (the collection methodology is detailed in Section 3.2 in my dissertation), therefore, I calculated cluster configurations for all potential k values. I computed the $s(k)$ metric for all cluster configurations in order to rank the different setups. As shown in Table 1, the best $s(k)$ values of the calculated cluster configurations barely differ, however, the corresponding k values have a wide range, making it unclear which setup to choose. I also observed that several cluster heads have small TLSH differences when compared to other cluster heads, which suggests that clusters could be merged. As variants have a TLSH difference lower then 48 (the details of deriving this number can be found in Section 3.2.3 of my dissertation), cluster heads of different clusters should have a TLSH difference score higher than 48. With this requirement in mind, I looked at the cluster configurations and found that with TLSH thresholds ranging from 30 to 70, $k = 17$ achieves the best $s(k)$ value.

Statistics of the cluster configuration $k = 17$ is shown in Table 2. In this configuration, the calculated cluster diameters range from 180 to 1,038, the mean being 468. In the scenario of malware clustering based on TLSH similarity score, cluster diameter is interpreted as the largest TLSH difference between any two samples in the same cluster. Taking the TLSH difference threshold for variants of 48 into consideration, we can con-

clude that clusters in this setup contain very different samples; clusters should be split into smaller clusters.

OPTICS [1] identifies sparse and dense regions in the data set in order to create clusters. It takes two parameters, ϵ_{max} and $minPts$. ϵ describes the radius of an area, while $minPts$ is the minimum number of data points in that area. The algorithm dynamically calculates an $\epsilon \leq \epsilon_{max}$ values for data points such that data points have at least $minPts - 1$ samples in their ϵ radius. The algorithm also takes as input a precomputed distance matrix. OPTICS also has a built-in clustering algorithm, ξ, which clusters data points by detecting abrupt changes in the ϵ-values. This characteristic makes it favorable in my scenario as my data set has malware families with only a few samples, as well as families with thousands of samples.

We can specify an upper bound for ϵ_{max} as the maximum TLSH difference in the data set. However, selecting $minPts$ is a challenge without knowledge about the internal structure of the data set. To gain this knowledge, I ran OPTICS with different parameter setups: ϵ_{max} values were set to be 40, 50, 60 and 70, while $minPts$ was set to be 1, 2, 5, 10, 20, 40, 50, 70, 100, 150 and 200.

The resulting cluster configurations were again unsatisfactory. In all configurations, the number of unclassified samples is very high. Different values of ϵ_{max} do not seem affect this trait: setting $minPts$ to 2, $\epsilon_{max} = 40$ yielded 1,800 unclassified samples, while $\epsilon_{max} = 70$ resulted in 1,721 unclassified samples. The more I increased $minPts$, the more unclassified samples were returned. The configuration $\epsilon_{max} = 70$, $minPts = 200$ resulted in 6,934 unclassified samples, which is 68% of the data set. Such a high number of unclassified samples is disadvantageous in malware clustering because many samples would require additional analysis.

THESIS 2.2: *In response to the unacceptable performance of k-medoid and OPTICS, I proposed a new malware clustering algorithm in [C3] and showed its superior performance to both k-medoid and OPTICS.*

My proposed clustering algorithm was developed with the following requirements in mind. First, it has to cluster samples based on their binary similarity expressed as TLSH differences. Second, it has to be able to find even the smallest clusters in a varying density data set. The input data set may contain singleton clusters, i.e. samples dissimilar to every other sample; however, these must not be treated as noise because these are the most interesting samples for malware analysis. The resulting algorithm is discussed in Section 3.4 of my dissertation.

In order to evaluate the efficiency of the proposed clustering algorithm, I compared it against the results of both k-medoid and OPTICS. During evaluation, I took into consideration cluster diameters, the number of singleton clusters generated and two new measures of goodness.

The number of generated clusters and singleton clusters are shown in Table 3. k-medoid and OPTICS both generate considerably fewer clusters than my algorithm does, more in line with the number of malware families the data set contains. My algorithm

Table 3: Comparison of the clustering methods

	k-medoid	OPTICS	My algorithm
Number of clusters	17	13	392
Number of singletons	0	6,058	353

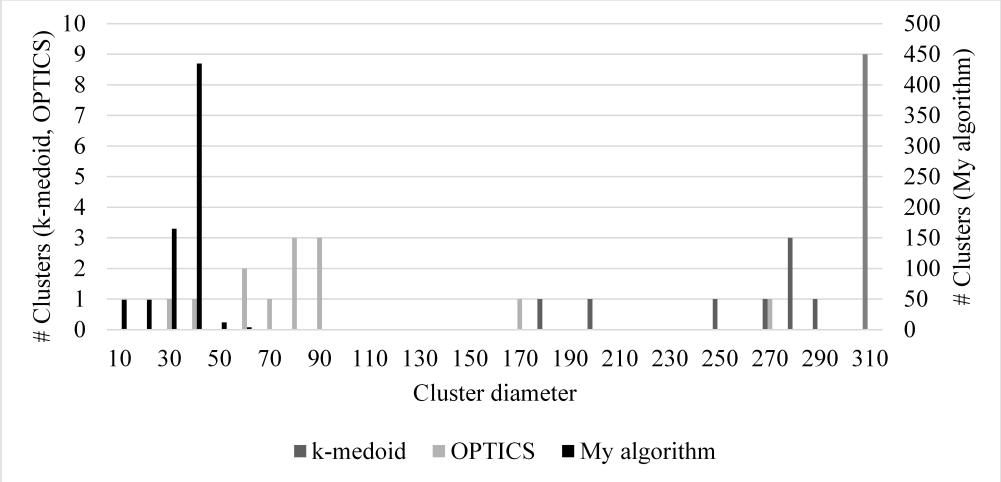

Figure 2: Cluster diameters produced by different clustering algorithms

generates 745 clusters, of which 353 are singletons, a negligible amount compared to OPTICS's performance.

The diameters of cluster configurations from all three algorithms are shown in Figure 2. Because my algorithm produced much more clusters than k-medoid and OPTICS, I use a different scale for the number of clusters in its case. The experiments detailed in my dissertation have shown that in order to detect variants of malware families, cluster diameters must be below 48. The figure shows that both k-medoid's and OPTICS's cluster diameters are too large to denote variants. The cluster configuration of my algorithm, however, is much closer to this threshold with 93.69% of my clusters having diameters below 50. As a result, my clusters are more likely to represent malware variants.

The first measure of goodness I present shows how "pure" a cluster is, i.e. how many samples of the cluster are of the family with the most samples in that cluster. This metric can only be computed for non-singleton clusters as clusters containing only a single sample automatically achieve the measure of 1. Figure 3 shows the algorithms' performance with respect to this measure of goodness. Cluster configurations of both k-medoid and OPTICS typically achieve ratios between 0.56 and 0.63. By contrast, of the 392 non-singleton clusters produced by my algorithm, 185 have ratios over 0.6, of which 103 outperform both OPTICS and k-medoid.

We also need to take into consideration that malware families share and/or copy features from each other. In response, the relaxation of my measure of goodness considers not only the family with the most samples in a given cluster but also families with

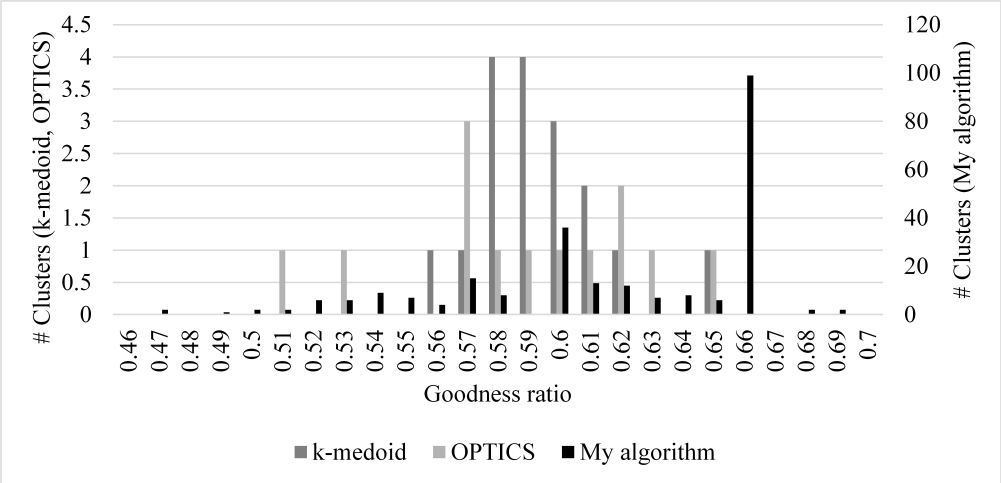

Figure 3: Goodness ratios produced by different clustering algorithms

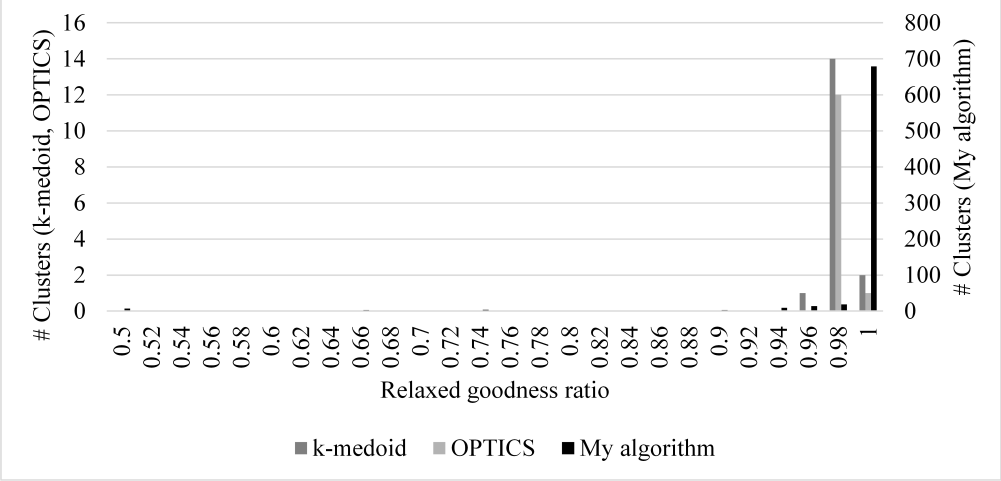

Figure 4: Relaxed goodness ratios produced by different clustering algorithms

which it is known to share features. Singleton clusters can be included, since non-singleton clusters have a chance of achieving the ratio 1. The relaxed measure of goodness ratios achieved by the compared algorithms are shown in Figure 4. The figure shows that even though both OPTICS and k-medoid achieve ratios above 0.95, my algorithm outperforms both with almost all clusters achieving the measure of 1. However, my algorithm produces clusters whose relaxed ratios are well below those achieved by k-medoid and OPTICS, likely the result of poor anti-virus labels.

2.3 Uncovering environmental requirements of malware

There exist malware clustering approaches that rely on features executed during samples' execution. However, attackers have developed malware in which the undocumented, potentially malicious features are executed only when specific conditions are met, for example, some inputs that satisfy pre-defined criteria are received. This behavior is known as *trigger-based behavior* and such inputs are called to as *trigger inputs*. The pre-defined criteria are hard-coded into the program in the form of checks and their semantic meaning can encompass all sorts of external requirements, e.g. specific system time or location, special text entered or messages received. Malware can evade in-depth analysis by scanning its environment and ceasing malicious activities if it finds signs of an analysis framework[5]. Trigger-based behavior also includes backdoors, a behavior prevalent in embedded firmware images [9], in which case access is granted, if a specific string is received as input.

Previous works in this field [5, 11, 17] have demonstrated the usefulness of symbolic execution [3, 24] to determine the conditions necessary to trigger hidden behaviors. Symbolic execution was originally developed to automate testing by analyzing execution paths and generating test cases, which lead execution down the analyzed execution path. In order to uncover the conditions related to trigger-based behavior, we need to analyze the program's interaction with its environment and the environment's influence on the program's behavior. If data from the environment is replaced with symbolic variables, symbolic execution can be used to analyze this interaction and obtain the hard-coded conditions. Solving these conditions give analysts concrete values that can be used for further analysis. However, using symbolic execution has a limitation: the more symbolic variables are introduced into the analysis, the more execution paths must be analyzed, leading to the *path explosion problem*. Previous work addressed this problem by considering only a subset of potential trigger condition types.

[5] https://www.fireeye.com/blog/threat-research/2011/01/the-dead-giveaways-of-vm-aware-malware.html Last visited: Jun 8, 2020

THESIS 3.1: *I presented a general approach for uncovering environmental conditions posed by stealthy malware. This approach is capable of considering all external data sources as trigger input types. I validated the applicability of the general idea at the source code level using real-world, open-source software samples in [C4][6]. The results demonstrate that if symbolic execution succeeds, environmental conditions guarding hidden behaviors can be extracted from programs. However, the path explosion problem remains a challenge.*

I improved upon existing work by considering all potential trigger condition types automatically. In order to deal with the greater number of symbolic variables and the resulting path explosion problem, my approach focuses on specific instances of malicious behavior. Therefore, my method requires the human analyst to provide a description of the malicious behavior in the form of a sequence of system calls with associated predicates: $(s_0, p_0), (s_1, p_1), \ldots, (s_n, p_n)$. Throughout the booklet, I refer to this sequence as the *target system call pattern* of the malicious behavior.

Modeling malicious behaviors using a sequence of system calls is a widely used approach in the malware analysis domain [19, 28, 14, 8, 13, 18] for a number of reasons. First, system calls constitute the primary communication channel between programs and the operating system. Many functionalities necessary for a program's execution are provided as services by the operating system which can be required via system calls. Second, the semantics of each system call are documented and available for interested parties. However, system calls alone are often insufficient to describe a particular behavior with enough granularity [7]. Therefore, I allow users to specify additional requirements for the invocation of system calls with predicates. Such requirements may include specifying what arguments should be passed to system calls, what the contents of the execution state's memory and/or registers should have, etc. While specifying target system call patterns requires expert knowledge, they need to be defined only once and can be reused later for many samples. In this regard, they are similar to other knowledge bases related to malware analysis, e.g., YARA rules[7]. In addition, individual low-level target system call patterns could be combined to build higher-level behavioral specifications similarly to the work of Lorenzo *et al.* [22].

The overview of the main idea is shown in Figure 5. I assume that the analyzed program is deterministic and interacts with the environment through either libraries or the operating system and its API (system calls). In real-life execution, the program would invoke multiple calls and a subset of the return values would be matched against the pre-defined criteria hard-coded into its logic. Only if the result of the comparison(s) were a match, would the program execute the potentially malicious behavior. In order to analyze this interaction, the return values of those calls that return data from external sources must be replaced with fresh symbolic variables.

I implemented this methodology as a prototype using the GNU and LLVM toolchains.

[6] Thorsten Tarrach supervised my work for AIT Austrian Institute of Technology GmbH.
[7] https://virustotal.github.io/yara/ (Last visited: Mar 5, 2021

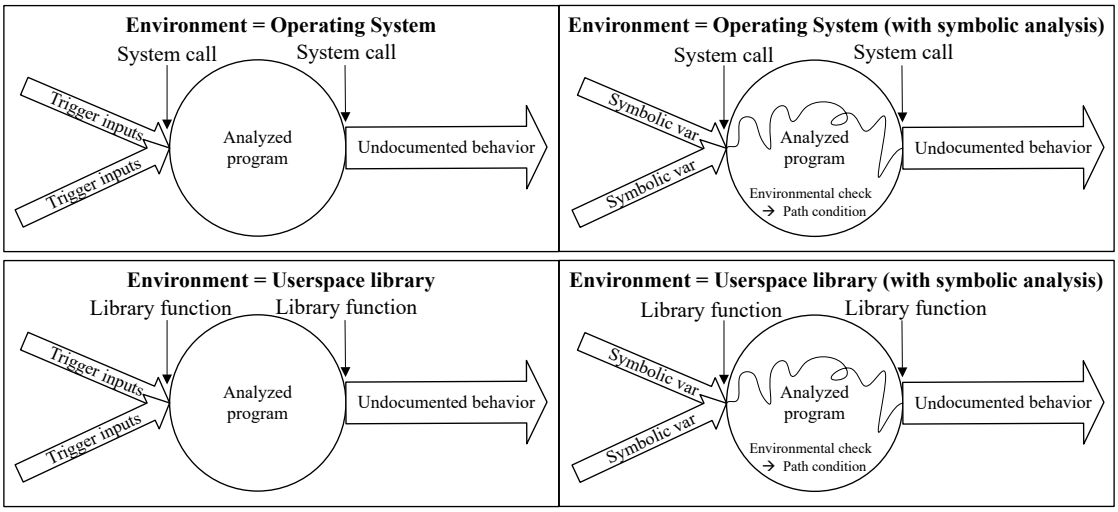

Figure 5: Using symbolic execution for recovering environmental conditions guarding malicious behavior

Table 4: Results for uncovering hidden malicious behavior in open-source software

Sample Name	**Completed paths**	**Generated test cases**	**Detected**
cd00r	1,299	5 (1 highlighted)	Yes (1/1)
giardia	48	4 (1 highlighted)	Yes (1/1)
osx-ping-backdoor	212,754	122 (2 highlighted)	Yes (1/2)
portknockd	11,902,399	1	No
portknocking	39,077	8	No

My prototype automatically generates symbolic summary functions based on a list of function names and semantic data about how to introduce fresh symbolic values. The prototype uses KLEE [6] as a mixed concrete and symbolic execution tool. My implementation signals KLEE to output highlighted test cases at potentially malicious instructions using the `klee_assert()` function.

To evaluate my proposed approach, I collected open-source software from GitHub using keyword search for the terms “backdoor”, “logic bomb”, “time bomb” and “port-knock”. All collected samples are written in C and implement some form of trigger-based behavior. For the evaluation, I used a virtual machine with 4 CPUs and 10 GB memory. The virtual machine ran Ubuntu 14.04.5 LTS. I set the maximum memory available to KLEE to 8 Gb and used its default path selection strategy.

The results are summarized in Table 4. While KLEE explored many paths in the samples, I 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. In three out of five projects, KLEE generated true positive test cases, i.e., trigger inputs for malicious execution paths. In the remaining cases, the limitations of KLEE lead to unsuccessful

experiments. The results also shows that my proposed approach suffers heavily from the path explosion problem.

THESIS 3.2: *In order to tackle the path explosion problem, I proposed a new analysis method relying on directed symbolic execution [21] to guide analysis towards selected program points, for example, potentially malicious behaviors in binaries. I implemented the proposed analysis method in angr [25] and evaluated it on both artificial and real malware samples in [C5][8]. My experiments proved that it is possible to obtain the environmental conditions required to trigger specific program points. In addition, analysis has a reasonable performance considering the complexity of the analyzed samples and the generality of my approach.*

The pseudocode of the proposed path selection strategy can found as Algorithm 2 in my dissertation. The details of the algorithm are discussed in Section 4.4. My proposed analysis consists of three techniques:

1. a path selection strategy consisting of three levels to prioritize available execution paths,
2. symbolic summary functions capturing the behavior of invoked system calls in order to introduce a model of environmental data to the analysis, and
3. a mechanism for automatically finding target program points that can advance the execution path's progress.

Analysis starts from the entry point of the program and while there are execution states to be analyzed, we select the most promising state(s) according to the path selection strategy. The selected state(s) are symbolically analyzed using mixed concrete and symbolic execution. In order to model the environment, custom symbolic summary functions must be supplied.

I implemented a prototype in angr and evaluated the performance of the prototype on two malware samples. Both pieces of malware poses several challenges for analysis. First, due to my assumptions and their implementations, a large number of execution paths are available for analysis, the sources of which are the following:

1. Environmental data: My pieces of malware relies on the system time, process IDs and communication over the network. As I assume no prior knowledge about its functionality, my analysis has to analyze all those inputs using symbolic variables, leading to many branches.
2. String handling: My pieces of malware processes network input using standard libc functions such as `strlen` and `strcasecmp`. These functions typically loop over the string character by character. As their inputs are returned from the kernel, my analysis must consider each of the characters a symbolic variable. Such loops are known to contribute to the path explosion problem.

[8] Thorsten Tarrach supervised my work for AIT Austrian Institute of Technology GmbH.

Table 5: Solutions to the symbolic characters which trigger the DoS attack

Character index	Possible solutions
0	`d` or `D`
1	`o` or `O`
2	`s` or `S`
3	SPACE
4	any character
5	any character
6	any character
7	any character
8	SPACE
9	terminating `NULL`

3. Infinite loop: My pieces of malware is implemented to run in an infinite loop, continuously listening for messages from the command and control server and trying to reconnect in case of communication failure. As a result, exploring all execution paths cannot be done in a finite amount of time.

My artificial malware sample has multiple instances of trigger-based behavior. It first checks whether it executes in a virtual environment and exits, if it determines it does. If no virtual environment is detected, my piece of malware connects to its hard-coded command and control server and leaks several parameters of the host, including operating system and kernel versions, total and available memory, and the number of running processes. It also accepts commands from the command and control server. The commands can cause it to launch a TCP PUSH+ACK-type denial-of-service attack against the specified target, or to kill itself and all spawned child processes.

An execution path, which detected a virtualized environment, was found in ca. 7 minutes (437.17 seconds) and required a total of 5.96 GB of RAM. In order to reach the necessary system calls, analysis generated a total of 71 execution paths before finding one that detects the virtual environment and exits. The path condition of this execution path contains constraints which show the necessary ASCII values encoding the virtual environment: `"Q"`, `"E"`, `"M"`, `"U"`.

Finding an execution path which leaks system parameters completed in ca. 9 minutes (568.09 seconds) and used 5.8 GB of RAM. There were 4 execution paths to which my path selection strategy assigned equal priorities; their analysis uncovered the necessary behavior in the same iteration.

My analysis was able to find execution paths that executes a denial-of-service attack in ca. 5 hours (299 minutes) and required 41.4 GB of RAM. The path conditions shed light on the necessary trigger, as shown in Table 5. I manually fed the concrete input returned by the solver to my piece of malware and found that it indeed triggers the execution of the denial-of-service attack.

Table 6: Runtime performance onreal malware

Stage	Runtime (hh:mm:ss)
Control-flow graph generation and extension	0:10:42
Simulation of execution paths	19:08:54
Shortest distance calculation	8:05:44
Other management tasks	5:05:11

I also used a publicly available Kaiten sample[9] during evaluation. I selected one of the functions launching denial-of-service attacks (`tsunami` in the source code) as the target behavior.

Table 6 shows the performance of the prototype implementation on the Kaiten binary sample. The execution time of a single run consists of four components:

1. generation and extension of the control-flow graph,
2. simulating execution paths,
3. ranking execution states (based on shortest distance), and
4. other management tasks, e.g., logging events, checking if the target was reached, etc.

The measured execution time of my analysis was 32.5 hours. Most of the time was spent with either simulating execution paths or calculating shortest distances.

The execution time of analyzing execution paths can be accredited to the logic of the sample. During the tests, there were addresses whose analysis took hours for mixed concrete and symbolic execution. These addresses were part of libc, including `rand` and multiple string manipulating functions whose analysis involved computations with complex symbolic values. `rand` is used by the sample to generate random 1-character-long strings for communication with the C&C server. While the generated string has to be analyzed in order to reach the target system call, its value does not matter: the symbolic string representing network input either matches it, or it does not. Therefore, I replaced `rand` with angr's built-in symbolic summary function and used a fresh, unconstrained symbolic variable to represent its result. However, the results of string manipulations contribute directly to the execution path leading towards the selected target behavior: they affect how long the symbolic string representing network input is and what constraints are placed on its characters. Therefore, I did not influence the execution of string manipulations and settled for the increased execution time.

All previous experiments were performed on a machine with two Xeon E5-2680 CPUs of 10 cores each, running at 2.8 GHz. The machine has 378 GB of RAM available. Note that angr is not multithreaded and uses only a single core. I also restricted angr to 330 GB of memory.

[9] https://packetstormsecurity.com/files/25575/kaiten.c.html (Last visited: Jan 8, 2021)

The path conditions of execution states reaching the targeted (sequence of) program points have constraints associated with them whose interpretation is of varying difficulty. Constraints like `<Bool socket_retval_23127_32 == 0x3>` are easy to interpret with knowledge about the semantics of the system call whose model created them. As its name suggest, `socket_retval_23127_32` is the symbolic variable introduced when the `socket` system call is invoked. The two numbers are appended by angr: the first is a unique identifier, while the second is the length of the variable in bits. The return value of `socket` in case of success is a file descriptor (positive integer) and in case of failure, it is -1. Given that the right-hand side of the equation is positive, we can deduce that the `socket` system call had to be completed successfully.

The human interpretation of other constraints, however, is quite challenging due to their complexity. For example, the Kaiten sample sets an upper limit of 4096 on the number of characters it reads from a socket with one call. As my symbolic summary function of `recv` has an upper bound of 10 for the number of characters to be returned, it returns a string with at most 10 characters. The malware sample then invokes multiple string manipulating functions, which loop over the string character by character. The corresponding binary instructions are conditional in many cases, which means that in real life, the CPU would execute them only if necessary. During analysis, however, one of their operands is a symbolic character and therefore, they cannot be skipped. Instead, when possible, their results are encoded into `If-Then-Else` structures in the path condition: if the flag evaluates to true, then the result is the `Then` value, else the `Else` value. These structures can be nested into each other, leading to constraints whose evaluation is tedious manually. In such cases, the constraint solver can be used to calculate satisfying value assignments, giving concrete inputs to trigger the targeted behavior. Another challenge in interpreting constraints related to input strings is that the constraint does not include characters. Instead, (in)equality checks use the characters' numeric ASCII values. However, the naming convention of symbolic variables tells the analysts at which system call a variable was introduced to analysis. Thereby the analyst gains semantic information that the symbolic variable's value can be interpreted as characters.

2.4 Proactive security for embedded IoT devices

My results regarding the threat landscape demonstrate that malware infected embedded IoT devices can endanger various application domains. Therefore, these devices must conform to security requirements, which is achieved using security controls and mechanisms. However, there is a wide range of vulnerabilities that need to be addressed. Doing so individually for each vulnerability is a tedious task, unfortunately, which does not scale well. Therefore, I turned my attention to proactive security. In contrast to reactive security, which has the goal of detecting and reacting to attacks and compromises, proactive security mechanisms anticipate attacks and put measures in place to prevent compromises from happening.

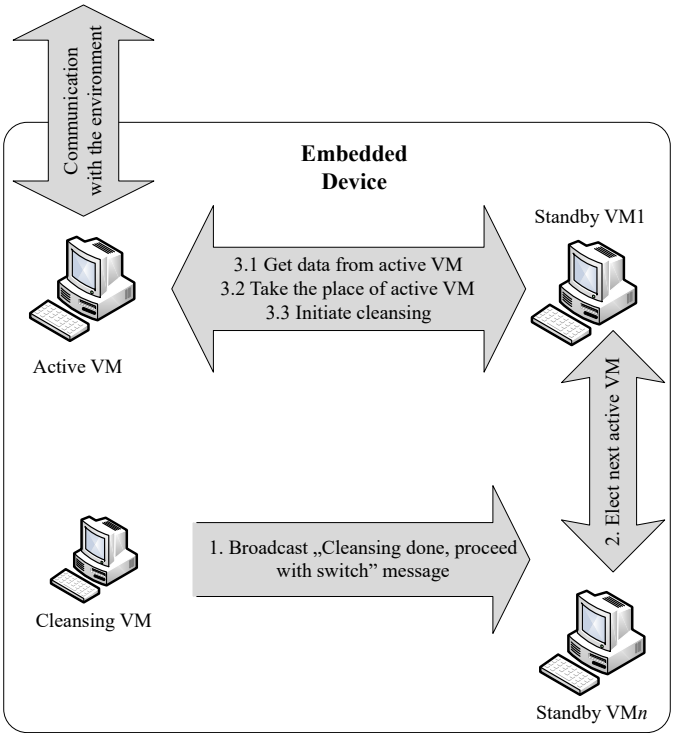

Figure 6: High-level overview of RoViM

THESIS 4: *I designed a new mode of operation, called RoViM, for embedded IoT devices in [C6][10], which allows them to periodically restore themselves to a compromise-free state. I formally verified the design using* Uppaal *[4] and proved reachability, liveness, and safety properties. I also implemented a prototype of RoViM as an IPsec [16] gateway and measured its performance. My measurements showed that my proposed new mode of operation does not have a significant impact on user experience.*

The basic idea of RoViM is to have multiple virtual machines on an embedded device and periodically rotate between these virtual machines. Before the high-level overview of the system can be discussed, some definitions must be introduced. The virtual machine that is connected to and communicating with the outside world, and that performs the task of the embedded device will be called *active* virtual machine. *Standby* virtual machine(s) provide redundancy and are on cold standby, waiting to replace the active virtual machine. The standby virtual machine that will become the active virtual machine in the rotation is called the *next active* virtual machine. The *cleansing* virtual machine previously acted as the active virtual machine and is being restored to its compromise-free state. In practice, the compromise-free state can be a snapshot taken before the deployment of the embedded device. Rotations between virtual machines happen periodically.

[10] Zhendong Ma supervised my work for AIT Austrian Institute of Technology GmbH.

Figure 6 shows one rotation with the high-level interaction needed for a single cycle to complete. The unnumbered arrow between the active virtual machine and the outside world highlights that the communication between the active virtual machine and the outside world is not disrupted by the rotations. Rotations are triggered periodically by the cleansing virtual machine after it is restored to a compromise-free state. The trigger itself is a broadcast message to all standby virtual machines, instructing them to begin the second phase of the rotation.

In the second phase, a standby virtual machine becomes the next active virtual machine. Depending on the number of standby virtual machines used, two cases must be considered. If there is only one standby virtual machine (apart from the previously cleansed virtual machine), that virtual machine will automatically be the next active virtual machine. If there are multiple standby virtual machines, they must agree on which standby virtual machine should become the next active virtual machine. This problem translates to the well-known leader election problem [20].

To become the active virtual machine, a three-subphase interaction is needed between the active and the next active virtual machines. A high-level description of the three subphases is presented here, more details are discussed in Section 5.3 in my dissertation.

1. The next active virtual machine must acquire all required data to perform the task of the embedded device correctly. As the active virtual machine is connected to the outside and may be compromised, the data on it may become corrupted or malware can be installed. The designed system can be extended to ensure that no malicious content is propagated to other virtual machines via input validation of the application data. In addition, while the data from the active virtual machine is being transmitted, the active virtual machine must make no changes to the application data. Otherwise, the application running on the next active virtual machine and the entities in the outside world would lose synchronization. In a sense, time must freeze for the application but this may be against the availability requirements of the application. Therefore, the implementation must specify a time limit during which the next active virtual machine can take the place of the active virtual machine. If the next active virtual machine does not succeed within that time limit, the rotation should be aborted.

2. The next active virtual machine must notify all nodes on the local network to route packets currently destined to the active virtual machine to the next active virtual machine instead.

3. The next active virtual machine must initiate the restoration of the active virtual machine into a compromise-free state. The active virtual machine is connected to the outside world and may be compromised. We can assume that restoration to a compromise-free state is against the interests of the attacker. Therefore, the next active virtual machine must force cleansing. One such cleansing procedure can be reverting the active virtual machine to a snapshot taken before the deployment of the embedded device.

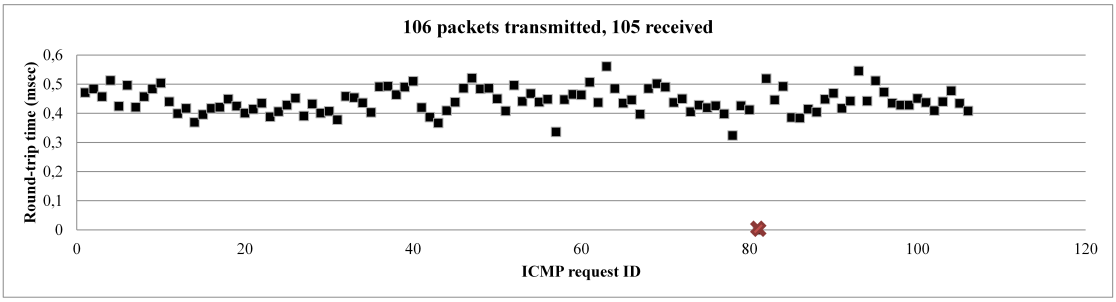

Figure 7: Round-trip times of individual ICMP requests and corresponding replies

In order to limit packet loss, packets can be buffered during the interaction and can be later retransmitted.

The outcome of the three-phase interaction should be that the virtual machines reach a global state in which either

- the switch happened without errors and the elected standby virtual machine replaced the active virtual machine, or
- the global state before the interaction is restored in case of errors and the active virtual machine is still in the active role.

It is also important to check whether the proposed third phase can lead to virtual machines having inconsistent states and whether the protocol implementing that phase has any deadlocks. To answer these questions, I subjected the proposed protocol to formal verification using Uppaal [4]. The formal verification was not aimed to finding security issues, but to check the correctness of the protocol with respect to functionality. Uppaal proved reachability, liveness, and safety properties of the protocol. The details of this analysis can be found in Section 5.4 in my dissertation.

I evaluated the performance of the IPsec gateway prototype with respect to packet loss and user experience. For the measurement concerning packet loss, I used ICMP requests and replies: I executed a `ping` command on a `Client` virtual machine aimed at a `Server` virtual machine (their secure communication is provided via an IPsec tunnel). Meanwhile, I triggered one cycle of rotation in the IPsec gateway manually.

The results of this experiment are shown in Figure 7. The rotation was triggered when ICMP request #81 was sent from the `Client`. The round-trip times of individual ICMP requests and replies show that no significant latency was introduced to the packet flow. However, the `Client` suffered a packet loss for ICMP request #81. The log files on the prototype implementation provided more insight into the issue: the active virtual machine could not process ICMP request #81 fast enough for the rotation, which was not forwarded to the next active virtual machine. Running the measurement multiple times resulted in the same outcome.

The second experiment I conducted aimed at finding changes in the user-experience. Therefore, I downloaded a 500 MB file from the `Server` to the `Client`. As in the

Table 7: Continuous TCP packet flow statistics with and without rotation

	Without rotation	With rotation
Transmission time	23.680 s	23.684 s
Duplicate IP address configured (172.16.5.254)	0	2
Retransmissions	55	315
Out-of-order segments	32	39

previous test, the rotation at the RoViM-enabled IPsec gateway prototype was triggered manually. The HTTP protocol used to download the file uses TCP, therefore, the packet flow also provided insight into how the latency introduced by the three-phase interaction influences TCP. I ran Wireshark at the `Server` to observe the packet flow and gather statistics.

The results are shown in Table 7. The TCP connection between the `Client` and the `Server` did not break during the rotation. As expected, the rotation introduced latency to the transmission, but the transmission time increased with only 4 ms, which does not influence the user-experience.

The next active virtual machine's network interface had the same IP address as the active virtual machine's network interface. At the `Server`'s side, it seemed that while the IP address of the gateway did not change, its MAC address did. Therefore, Wireshark issued a `Duplicate IP address configured` warning. This warning was present twice in the packet flow due the IPsec gateway's rotation.

The rotation introduced a significant increase in retransmissions for TCP. In my case, the network throughput in the test environment was very high until rotation. This was because, first, there is no other source of traffic, and second, the test environment is also virtual and the virtual machines are simply ports from the host's point of view. The retransmission timeout calculated by Wireshark was 0.25 s before the rotation. Then, the three-subphase interaction occured and suddenly, all incoming packets were buffered at the active virtual machine. Acknowledgements for segments did not arrive within timeout, making the `Server` believe that some kind of network error had occurred. Retransmissions took a total of 6.556 ms to complete.

The increase in out-of-order segments was also caused by the rotation. When the next active virtual machine had its interfaces up, new packets were allowed to flow and buffered ones were retransmitted by the IPsec gateway. In case of this experiment, this resulted in buffered acknowledgements for previous segments, causing the segment flow to become out-of-order.

Based on the discussed issues, it seems that the optional buffering feature of the rotation only hinder the performance of TCP. The experiment was repeated with the feature disabled and as a result, TCP needed only 5.113 ms to retransmit the buffered segments.

List of Publications

Conference and Workshop Publications

[C1] Papp, D., Ma, Z., and Buttyán, L. Embedded systems security: Towards an attack taxonomy. In *Mesterpróba 2015* (2015), pp. 29–32.

[C2] Papp, D., Ma, Z., and Buttyan, L. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)* (2015), pp. 145–152.

[C3] Bak, M., Papp, D., Tamás, Cs., and Buttyán, L. Clustering IoT malware based on binary similarity. In *6th IEEE/IFIP Workshop on Security for Emerging Distributed Network Technologies (DISSECT)* (2020), NOMS '20, pp. 1–6.

[C4] Papp, D., Buttyán, L., and Ma, Z. Towards semi-automated detection of trigger-based behavior for software security assurance. In *4th International Workshop on Software Assurance (SAW 2017)* (New York, NY, USA, 2017), ARES '17, Association for Computing Machinery.

[C5] Papp, D., Tarrach, T., and Buttyán, L. Towards detecting trigger-based behavior in binaries: Uncovering the correct environment. In *Software Engineering and Formal Methods* (Cham, 2019), P. C. Ölveczky and G. Salaün, Eds., Springer International Publishing, pp. 491–509.

[C6] Papp, D., Ma, Z., and Buttyán, L. RoViM: Rotating virtual machines for security and fault-tolerance. In *EMC2 Summit at CPS Week* (2016).

[C7] Papp, D., Kócsó, B., Holczer, T., Buttyán, L., and Bencsáth, B. ROSCO: Repository Of Signed COde. In *Virus Bulletin Conference* (Prague, Czech Republic, 2015).

[C8] Juhász, M., Papp, D., and Buttyán, L. Towards secure remote firmware update on embedded IoT devices. In *12th Conference of PhD Students in Computer Science* (2020).

[C9] Tamás, C., Papp, D., and Buttyán, L. SIMBIoTA: Similarity-based malware detection on IoT devices. In *6th International Conference on Internet of Things, Big Data and Security* (2021), IoTBDS '20.

Journal Publications

[J1] Papp, D., Tamás, K., and Buttyán, L. IoT hacking–a primer. *Infocommunications Journal 11*, 2 (2019), 2–13.

[J2] Papp, D., Zombor, M., and Buttyán, L. TEE based protection of cryptographic keys on embedded IoT devices. In *Annales Mathematicae et Informaticae*

(Special Issue on the Conference on Information Technology and Data Science) (2020). In press at the time of writing (May 8, 2021).

[J3] Nagy, R., Németh, K., Papp, D., and Buttyán, L. Rootkit detection on embedded IoT devices. In *Acta Cybernetica* (2021). In press at the time of writing (May 8, 2021).

References

[1] Ankerst, M., Breunig, M. M., Kriegel, H.-P., and Sander, J. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record* (1999), vol. 28, ACM, pp. 49–60.

[2] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., and Zhou, Y. Understanding the Mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, Aug. 2017), USENIX Association, pp. 1093–1110.

[3] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. 51*, 3 (2018).

[4] Behrmann, G., David, A., and Larsen, K. G. A tutorial on uppaal. In *Formal methods for the design of real-time systems*. Springer, 2004, pp. 200–236.

[5] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 2008, pp. 65–88.

[6] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[7] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2012), ISSTA 2012, Association for Computing Machinery, p. 122–132.

[8] Canzanese, R., Mancoridis, S., and Kam, M. System call-based detection of malicious processes. In *2015 IEEE International Conference on Software Quality, Reliability and Security* (Aug 2015), pp. 119–124.

[9] Costin, A., Zaddach, J., Francillon, A., and Balzarotti, D. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 95–110.

[10] Cozzi, E., Vervier, P.-A., Dell'Amico, M., Shen, Y., Bigle, L., and Balzarotti, D. The tangled genealogy of IoT malware. In *Annual Computer Security Applications Conference (ACSAC2020)* (Austin, USA, 2020). At the time of writing (Nov 18, 2020), the paper has been accepted to the conference but not yet published. The authors made the paper available to the public.

[11] Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., and Vigna, G. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 377–396.

[12] Gibert, D., Mateu, C., and Planes, J. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications 153* (2020), 102526.

[13] Gupta, S., Sharma, H., and Kaur, S. Malware characterization using windows api call sequences. In *Security, Privacy, and Applied Cryptography Engineering* (Cham, 2016), C. Carlet, M. A. Hasan, and V. Saraswat, Eds., Springer International Publishing, pp. 271–280.

[14] Hubballi, N., Biswas, S., and Nandi, S. Sequencegram: n-gram modeling of system calls for program based anomaly detection. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)* (Jan 2011), pp. 1–10.

[15] Kaufman, L., and Rousseeuw, P. J. *Finding groups in data: an introduction to cluster analysis*, vol. 344. John Wiley & Sons, 2009.

[16] Kent, S., and Seo, K. Security architecture for the internet protocol, 2005. Also known as RFC4301.

[17] Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C. Rozzle: De-cloaking internet malware. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 443–457.

[18] Lee, T., Choi, B., Shin, Y., and Kwak, J. Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient. *The Journal of Supercomputing 74*, 8 (2018), 3489–3503.

[19] Liu, M., Xue, Z., Xu, X., Zhong, C., and Chen, J. Host-based intrusion detection system with system calls: Review and future trends. *ACM Comput. Surv. 51*, 5 (Nov. 2018).

[20] Lynch, N. A. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[21] Ma, K.-K., Yit Phang, K., Foster, J. S., and Hicks, M. Directed symbolic execution. In *Static Analysis* (Berlin, Heidelberg, 2011), E. Yahav, Ed., Springer Berlin Heidelberg, pp. 95–111.

[22] Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., and Mitchell, J. C. A layered architecture for detecting malicious behaviors. In *Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2008), R. Lippmann, E. Kirda, and A. Trachtenberg, Eds., Springer Berlin Heidelberg, pp. 78–97.

[23] Oliver, J., Cheng, C., and Chen, Y. Tlsh–a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop* (2013), IEEE, pp. 7–13.

[24] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy* (May 2010), pp. 317–331.

[25] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 138–157.

[26] Ucci, D., Aniello, L., and Baldoni, R. Survey of machine learning techniques for malware analysis. *Computers & Security 81* (2019), 123 – 147.

[27] Ye, Y., Li, T., Adjeroh, D., and Iyengar, S. S. A survey on malware detection using data mining techniques. *ACM Comput. Surv. 50*, 3 (June 2017).

[28] Yu, B., Fang, Y., Yang, Q., Tang, Y., and Liu, L. A survey of malware behavior description and analysis. *Frontiers of Information Technology & Electronic Engineering 19*, 5 (2018), 583–603.