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

Improved security and protection from malware for embedded IoT devices

Ph.D. Dissertation
of
Dorottya Futóné Papp

Supervisor: Levente Buttyán, habil. Ph.D.

Budapest, Hungary
2020

Abstract

The field of embedded devices is changing rapidly. While these devices have originally been developed to perform specific tasks, they are now increasingly connected to the Internet. This new lifephase of the Internet, in which there are more connected devices than people, is called the Internet of Things. Consequently, embedded devices are increasingly called embedded IoT devices.

On the one hand, Internet connectivity enables new and innovative application areas, including smart home appliances, automated traffic management in smart cities, and remotely monitored personal healthcare devices. On the other hand, Internet connectivity is a new attack surface for embedded devices which need protection. Indeed, there have been a rising number of attacks against and security incidents involving embedded IoT devices. The threats these devices face include malware, data theft, illegitimate access, and hijacking, among others. Therefore, there is a growing need to enhance embedded IoT devices with cybersecurity capabilities.

In this dissertation, we investigate the security of embedded IoT devices from multiple aspects. First, we study the threat landscape and derive an attack taxonomy to classify common attack scenarios. We draw two conclusions from this study, namely, that malware is a significant threat in the case of embedded IoT devices and that the vulnerabilities of these devices form a diverse set.

In order to tackle the issue of malware, we use malware clustering to find groups of similar malware, thereby reducing the workload of analysts tasked with understanding individual samples. The rationale behind malware clustering is that samples may share similar features and knowing the behavior of a sample can be used to approximate the features of similar samples. Specifically, we use binary similarity hashes to capture byte-level similarity between malware samples. We show that existing clustering algorithms perform poorly with this feature, hence we propose a new clustering algorithm with superior performance.

We also investigate a type of stealthy malware, which exhibits malicious behavior only when it receives specific inputs from its environment, e.g., specific command line inputs or network packages. Malware samples with this capability are especially challenging to analyze because the analyst has no knowledge about the necessary inputs. Therefore, we propose a new analysis method that can uncover environmental conditions hard-coded into malware samples to allow analysts to construct the right environment for observing potentially malicious behavior. We evaluate our proposed method on both artificial and

real malware samples and show that it has acceptable performance.

Based on our investigation into the threat landscape, we know that there is a diverge set of vulnerabilities to be addressed. We believe that doing so individually is not scalable enough to match the pace with which these devices are deployed in our daily lives. Therefore, we propose a new mode of operation, called RoViM, for embedded IoT devices, which accepts the possibility of compromises but allows devices to periodically cleanse themselves, restoring a compromise-free state. We formally verify the design of the proposed mode of operation and show that the interactions leading up to cleansing satisfy reachability, liveness, and safety properties. We also investigate the impact that RoViM has on user experience with a prototype implementation and show that the latency introduced by cleansing has no significant impact.

Kivonat

A beágyazott eszközök területe gyorsan változik. Míg ezeket az eszközöket eredetileg jól meghatározott feladatok ellátására fejlesztették ki, ma már egyre több ilyen eszközt csatlakoztatnak az internethez. Manapság az interneten több a csatlakoztatott eszköz, mint az emberi felhasználó, ezért gyakran hivatkozunk rá a dolgok internete (Internet of Things, röviden IoT) elnevezéssel. Következésképpen a beágyazott eszközöket is egyre inkább beágyazott IoT-eszközöknek nevezzük.

Az internetkapcsolat egyfelől új és innovatív alkalmazási területeket tesz lehetővé, beleértve az intelligens háztartási készülékeket, az intelligens városok automatizált forgalomirányítását és a távfelügyelhető személyes egészségügyi készülékeket. Másfelől viszont az internetkapcsolat egy új támadási felület is a beágyazott eszközök számára, amely védelmet igényel. Valóban, egyre több támadás és biztonsági incidens köthető beágyazott IoT-eszközökhöz. Az ilyen fenyegetések közé tartoznak többek között a kártékony kódok, az adatlopás, a törvénytelen hozzáférés és az eltérítés. Ezért egyre nagyobb szükség van a beágyazott IoT-eszközök biztonsági képességeinek fejlesztésére.

Ebben a disszertációban több szempontból vizsgáljuk a beágyazott IoT-eszközök biztonságát. Először a fenyegetéseket tanulmányozzuk, és levezetünk egy új támadási taxonómiát, amit a gyakori támadási forgatókönyvek rendszerezéséhez használunk. A rendszerezett támadási forgatókönyvekből két következtetést vonunk le, nevezetesen azt, hogy a kártékony kódok jelentős veszélyforrást képeznek a beágyazott IoT-eszközök esetében, és hogy ezen eszközök sebezhetőségei változatos formákat ölthetnek.

A kártékony kódok kérdésére reagálva klaszterezéssel azonosítjuk hasonló kártékony kódok csoportjait, ezáltal csökkentve az egyedi minták megértésével megbízott elemzők munkaterhelését. A kártékony kódok klaszterezésének alapötlete az, hogy a minták hasonló tulajdonságokkal bírhatnak, és egy minta viselkedésének ismerete felhasználható a hasonló minták jellemzőinek közelítésére. Konkrétan bináris hasonlósági hash algoritmusokat használunk a kártékony kódok mintái közötti bájtszintű hasonlóság mérésére. Megmutatjuk, hogy a meglévő klaszterezési algoritmusok gyengén teljesítenek, amennyiben ezzel a tulajdonsággal reprezentálunk egy-egy mintát. Ezért egy új, jobb teljesítményű klaszterező algoritmust javasolunk.

Olyan rejtőzködő kártékony kódokat is vizsgálunk, amelyek csak akkor mutatnak rosszindulatú viselkedést, ha meghatározott bemeneteket kapnak a környezetükből, pl. speciális parancssori bemeneteket vagy hálózati csomagokat. Az ilyen képességű kártékony kódok mintáinak elemzése különösen nagy kihívást jelent, mivel az elemző nem is-

meri a szükséges bemeneteket. Ezért egy új elemzési módszert javasolunk, amely feltárja a kártékony kódok mintáiba kódolt környezeti feltételeket, hogy az elemző megfelelő környezetet állíthasson össze a potenciálisan rosszindulatú viselkedés megfigyeléséhez. A javasolt módszert mind mesterséges, mind valós rosszindulatú mintákon értékeljük, és megmutatjuk, hogy elfogadható teljesítményt nyújt.

Végül a beágyazott IoT-eszközök felé fordítjuk figyelmünket. A támadási forgatókönyvekből tudjuk, hogy a sérülékenységek különböző fajtáival kell foglalkoznunk. Meggyőződésünk, hogy ha ezt egyesével tesszük, akkor az nem skálázódik megfelelően ütemben ahhoz képest, amilyen ütemben jelennek meg ezek az eszközök a mindennapi életünkben. Ezért javasoljuk a beágyazott IoT-eszközök új működési módját, amit RoViM-nek nevezünk, és amely elfogadja a kompromittálódás lehetőségét, ám lehetővé teszi az eszközök számára, hogy periodikusan megtisztítsák önmagukat, helyreállítva egy kompromittálódás-mentes állapotot. Formálisan ellenőrizzük a javasolt működési mód terveit, és megmutatjuk, hogy a tisztításhoz vezető interakciók kielégítenek elérhetőségi, életképességi és biztonsági tulajdonságokat. Egy prototípuson keresztül azt is megvizsgáljuk, hogy milyen hatást gyakorol a RoViM működési mód a felhasználói élményre, és igazoljuk, hogy a tisztítás által okozott késleltetésnek nincs jelentős hatása.

Acknowledgement

I would like to thank my husband and family for all the support they showed me throughout my years as a PhD student. Their unwavering belief in my success was a great help and motivation to get though any and all challenges. It is thanks to them that my motto became "The question is not whether I can do it but by when."

I would also like to thank all members of the CrySyS Lab for the supporting workplace environment. They have created a community that values excellence and also cares about the needs of its members. This goes especially for Levente Buttyán, whom I see not just as my supervisor but my mentor as well. I hope I will be able to support and guide my future students the way he did.

I could work with very talented students as well, who helped me realize that my true passion within academic education is supervising student projects. I thank you all for the great moments we shared during research!

During my PhD, I had the opportunity to work with researchers from the Austrian Institute of Technology. This international collaboration provided me with both financial security and great learning opportunities. The discussions with Zhendong Ma and Thorsten Tarrach improved my communication and presentation skills, and their guidance undeniably contributed to the results in this thesis.

Finally, I gratefully acknowledge the financial support I received during these past years from the following projects, programmes, funding agencies and companies:

1. The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004)[1].
2. The SECREDAS project, which received funding from ECSEL Joint Undertaking as part of the European Union's Horizon2020 Research and Innovation Programme.
3. The H2020-ECSEL programme under grant agreement no. 692474 (AMASS).
4. The EU ARTEMIS project EMC^2 and the Austrian Research Promotion Agency (FFG).
5. 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]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.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 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 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 [5], 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 [33].

In this dissertation, we explore the security of embedded IoT devices. Specifically, we review the threat landscape of IoT devices with a new taxonomy for attack scenarios. The

proposed taxonomy is based on existing vulnerability data and highlights several aspects of attacks against embedded IoT devices. We evaluate the taxonomy on relevant records of the Common Vulnerabilities and Exposures[1] (CVE), a publicly available database of known vulnerabilities. The results highlight several areas in which improvements to the security of embedded devices are necessary.

One of the key insights gained from the analysis of CVE records is that malware is a major threat to embedded IoT devices. Therefore, we focus our attention on this threat and study the issue of malware from two aspects. First, we address the problem of malware clustering. Anti-virus companies rely on clustering techniques to group similar malware samples, thereby reducing the workload on human analysts and automated analysis tools tasked with analyzing samples. Given groups of similar samples, both human analysts and automated tools need to focus their attention on only those samples that have not been analyzed before and are not similar to any known malware. In this context, we study the applicability of an emerging binary similarity hash algorithm, TLSH by Trend Micro [91]. We find that existing clustering algorithms, namely k-medoid [60] and OPTICS [4], perform poorly with TLSH and consequently develop a new clustering algorithm with superior performance.

Second, we 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 both human analysts and automated tools. When analyzing malware, it is customary to execute samples in a controlled environment and observe their behavior for clues of malice. However, in the case of malware implementing trigger-based behavior, this approach does not yield results because the malware sample does not exhibit malicious behavior without the correct inputs. In order to overcome this challenge, we propose a new method to uncover environmental constraints guarding malicious behavior in an automated manner. We evaluate our proposed method on both artificial and real malware samples and find that it is indeed capable of recovering environmental constraints with acceptable performance.

Based on the results of analyzing CVE records, there is a wide range of vulnerabilities these devices can contain. Finding and patching these vulnerabilities individually is a tedious task, which does not scale well. Instead, we propose a new mode of operation, RoViM, for embedded IoT devices that allows them to restore themselves periodically to a known compromise-free state. We formally verify the proposed design and show that it satisfies reachability, liveness, and safety requirements. We also present a prototype implementation for RoViM and show that it does not affect user experience significantly.

The dissertation is structured as follows. Chapter 2 discusses the threat landscape of embedded IoT devices, including existing attacks and techniques for uncovering vulnerabilities in them. It also presents a new taxonomy for attack scenarios and evaluates it on records from the CVE database. Chapter 3 discusses malware clustering and binary similarity hashes. It also evaluates two well-known clustering algorithms, k-medoid [60] and OPTICS [4], and proposes a new clustering algorithm which achieves better performance

[1] https://cve.mitre.org/ (Last accessed: Dec 29, 2020)

than those two. Chapter 4 studies the problem of malware implementing trigger-based behavior and proposes a new method for uncovering the conditions under which specific malicious behaviors can be observed. The chapter also evaluates our proposed method on artificial and real malware samples. Chapter 5 presents RoViM, our proposed mode of operation for embedded IoT devices which allows them to periodically cleanse themselves of potential compromises. The chapter also discusses the formal verification of RoViM and the impact its prototype implementation has on user experience. Chapter 6 summarizes this dissertation and reviews the presented results. Because this dissertation addresses multiple and diverse aspects of the security of embedded IoT devices, we provide background on these various aspects in their respective chapters.

Chapter 2

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 the security engineering of 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 commonalities of the attacks, and the main vulnerabilities that they exploit.

In this chapter, we present a systematic review of existing threats and vulnerabilities. We focus on two sets of data, i.e., the exposures of attacks on embedded IoT systems in security conferences and literature, and the published vulnerabilities specific to embedded IoT systems. Based on the data, we derive an attack taxonomy to systematically identify and classify common attacks against embedded IoT devices and systems using such devices as building blocks. We evaluate the proposed attack taxonomy on relevant records from the CVE database. The results highlight several important aspects of the security of embedded devices and provides the basis for further research.

The chapter is structured as follows. Section 2.1 presents existing attacks against embedded IoT devices and systems.Section 2.2 presents and evaluates a new attack taxonomy, which can highlight potential attack scenarios for threat modeling. Section 2.3 concludes this chapter.

2.1 Examples for attacks against embedded IoT devices

We now list existing attacks against embedded IoT devices and systems and look into the attackers' capabilities and their implications. Although not comprehensive, in our view, the examples are very representative and cover a broad range of application domains such as industrial systems, communications, and consumer devices.

Keefe [61] presented a timeline of attacks for critical infrastructure. Noteworthy attacks date back to 1982 and the number of attacks have been increasing since 2001. Apa

et al. [6] presented vulnerabilities and possible exploits of key management in wireless devices. For example, one of the devices is shipped with a graphical user interface with default values to configure the device. The implementation of the interface generates a passphrase, which is later used to generate an AES key. However, the Pseudo-Random Number Generator is seeded by feeding the current time to the `srand()` function and the generator itself is the `rand()` function. As a result, the attacker can calculate the passphrase and the encryption key, and can intercept all communication on the target wireless network. Forner and Meixell [42] demonstrated remote attacks against SCADA devices using the ModBus protocol. The vulnerability they exploited is within the design of the protocol: it lacks encryption and authentication. As a result, devices can easily be exploited with a carefully crafted packet. RuggedCom devices can be attacked via hard-coded credentials in the operating system [28]. The default account in the system is necessary to support password recovery, therefore, it cannot be disabled. However, attackers knowing the MAC address can use this account to connect to the device and take full control of it.

Santamarta [107] presented multiple attacks against satellite communication systems originating from the ground segment. In one of the attack scenarios, the man-machine interface of the airplane onboard SATCOM unit requires administrator password for restricted configurations and control mechanisms. The generation algorithm uses the device serial number (which can be found printed on the device) and a hard-coded string, which makes it easy to guess the password. Thus, the attacker has access to all configurations and can disable critical parts related to the safety of the aircraft. Geovedi and Irayndi [44] implemented a rogue carrier for satellite systems. Their method allows the attacker to become an illegitimate user of services provided. First, the attacker must select its target, an artificial satellite. Then, the attacker points his antenna to the target and searches for unused, legal frequencies for clients. If such a frequency is found, the attacker is free to transmit and receive as he wishes. However, the attacker still has to avoid detection: he has to sniff packets sent by the operator to legitimate clients and do exactly as the operator packet asks. As stated in their talk, the method works because even if the satellite supports encryption, turning it on causes performance to drop significantly. As a result, operators usually turn it off. Costin and Francillon [29] investigated the Automatic Dependent Surveillance-Broadcast (ADS-B) protocol and presented practical attacks exploiting the vulnerabilities the protocol has: no authentication, no encryption and no challenge-response mechanisms. As a result, messages can be sniffed, spoofed or replayed. The attacker can confuse pilots and hinder them in performing their tasks.

Hernandez *et al.* [51] presented an attack against a smart home automation device, the Nest Thermostat. Pressing a button for 10 minutes on the device initiates a global reset. Afterwards, there is a small time window during which the device accepts code from connected USB sticks and uses that code for booting without any cryptographic checks on the code. An attacker can use this vulnerability to install an SSH server and access the home network of the user. However, physical access is needed to the device to launch the attack, therefore, the attacker either has to break into the house or

compromise the device during transport.

Checkoway *et al.* [25] presented physical and remote attack surfaces in cars. For example, the authentication protocol between the Telematics Unit and the center relies on a challenge-response mechanism. However, the random number generator is seeded with the same constant each time it is initialized. As a result, the attacker can replay a previously observed response packet to authenticate himself as the Telematics Call Center, gaining full control over the car.

Wireless home automation devices used for controlling electrical outlets can also be compromised[1]. The implementation of the Home Network Administration Protocol contains a buffer overflow vulnerability, which can be used to execute arbitrary code on the device. As the device controls the power outlet to all devices physically connected to it, the attacker can gain the ability to damage connected devices. The D-Link DIR-815 Wireless-N Dual Band Router contains a command injection vulnerability that allows the attacker to get remote access to the device[2]. The vulnerability lies with the packet parsing: strings inside backticks are considered commands and are executed on the router.

Cui *et al.* [34] discussed a case study of malicious firmware updates to a HP-RFU (Remote Firmware Update) LaserJet printer. The vulnerability, which enables this attack, comes from the fact that the printer has to accept printing jobs in an unauthenticated way (as dictated by the standard) and that the firmware is updated by printing to the memory. Thus, an attacker can send a printing job to the device, instructing it to update its firmware with the malicious code provided.

Costin and Francillon [30] discussed attacks against a fireworks control system. The protocol used by the system provides neither encryption, nor authentication, which allows the attacker to sniff packets and thus learn the addresses of each device. Now, the attacker might wait for the operator to `arm` the system, the attacker can immediately send the `digital arm` and `fire` commands. The system will immediately fire its pyrotechnics loads and may cause physical harm to the operator. The attack can be automated as well, since arbitrary Python code can be uploaded to the devices.

Hanna *et al.* [49] demonstrated multiple attacks against an automated external defibrillator. For example, the firmware upgrade software package shipped with the device has a buffer overflow vulnerability, which may result in arbitrary code execution. Another vulnerability is the use of CRC as a digital signature. Combining these two vulnerabilities allows the attacker to harm patients by setting shock protocols and shock strengths, or launch a cyberattack against the IT system in which the device is deployed.

[1] http://www.devttys0.com/2014/05/hacking-the-d-link-dsp-w215-smart-plug/ (Last visited: Jan 4, 2021)

[2] http://shadow-file.blogspot.hu/2013/02/dlink-dir-815-upnp-command-injection.html (Last visited: Jan 4, 2021)

2.2 New attack taxonomy for embedded IoT devices

Due to researchers' specific interests, the cost incurred in security testing, 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 study the information on vulnerabilities related to embedded systems, which we consider as the other side of the same coin. Our main information source is the Common Vulnerabilities and Exposures (CVE) database, 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 our research[3], the database contained more than 60,000 entries, not all related to embedded systems. We 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 is a set of attack classification criteria that serves as a basis for our attack taxonomy.

Analyzing CVE data was a major challenge in our work. The CVE database has more than 60,000 records, in which only a small part is relevant to embedded devices. CVE records do not contain meta-information that would make it easy to identify which records are related to embedded systems. Therefore, we applied heuristics to identify and extract the relevant subset. Specifically, we implemented a script that matched CVE records to a whitelist and a blacklist of keywords that we defined, and selected those entries whose textual description contains at least one word from our whitelist and did not contain any word from our blacklist. Our script identified 3826 relevant CVE records, which was still infeasible to read and analyze manually. In addition, the set of selected CVE records was quite biased in the sense that a large subset of the records was related to devices produced by a small number of embedded device manufacturers (e.g., 3,306 out of the 3,826 records were related to CISCO products).

Next, we randomly sampled the 3,826 CVE records based on the target mentioned at CVE Details[4] with the limitation that only 35 entries may relate to same target type (hardware, operating system or application). However, this resulted in the over-representation of some large companies (e.g., Cisco), so we had to set the limit of 9 for the number of CVE records from the same company. The result of the steps mentioned above was a sample set of 106 CVE records. We also added one CVE entry for which CVE Details did not provide information about the affected product. Finally, we manually analyzed all the 106 CVE records in the selected sample set, and identified appropriate criteria based on which the vulnerabilities in the CVE records can be classified.

Assuming that our selected sample set is representative, the identified classification criteria form the basis of a taxonomy of attacks against embedded devices. In order to check that indeed all 3,826 CVE records can be categorized according to the criteria derived from the analysis based on the 106 samples, we later created a script that classifies all 3,826 records according to our criteria in an automated way. Our script

[3] 2015, results were published in [C2].

[4] https://www.cvedetails.com/ (Last visited: Dec 11, 2020)

classified the majority of the records with no problem, and we were able to handle the few exceptions manually by refining our set of classification criteria. At the end of this process, we obtained a final set of criteria that allowed for the classification of all 3,826 CVE records related to embedded device vulnerabilities.

2.2.1 Related work

ENISA [39] maintains a list of existing incident taxonomies for general computer and IT systems. Among them is the common language security incident taxonomy developed at the Sandia National Laboratories, which divides an incident into attackers, tools, vulnerability, action, target, unauthorized results, and objective. Simmons *et al.* [112] proposes a cyber attack taxonomy that classifies cyber attacks into attack vector, operational impact, defense, information impact, and target, abbreviated to AVOIDIT. Common Attack Pattern Enumeration and Classification (CAPEC) [90] defines structured description of IT security attacks. It organizes attack patterns into 11 categories, such as data leakage attacks, resource depletion, injection, etc. Hansman and Hunt [50] proposed a four dimensional approach to attack taxonomy, including attack vector, target, vulnerabilities and exploits, and the possibility of having a payload or effect beyond itself. The information in each dimension is further described in several hierarchical levels of details. The MITRE ATT&CK framework[5] is a publicly available knowledge base of attack methods against enterprise IT systems and mobile systems observed in real-life. Attack methods are categorized into 14 dimensions with each dimension having multiple categories. Dimensions encompass the whole range of attack phases, from reconnaissance to lateral movement and impact.

There exist taxonomies for attacks against embedded systems, however, they do not cover the full spectrum of embedded IoT systems. Zhu *et al.* [135] provided a taxonomy of cyber attacks on Supervisory Control and Data Acquisition (SCADA) systems. Cyber attacks are classified according to their targets: hardware, software and the communication stack. The attacks on software are grouped into exploitation of embedded operating systems without privilege separation, buffer overflow, and SQL injection. The attacks on communication stack are classified into network, transport, and application layer, as well as the implementation of the protocols. Dessiatnikoff *et al.* [38] focused on potential attacks against onboard aerospace systems. Attacks are categorized into two major classes: attacks against core functions and against fault-tolerance mechanisms. For each subcategory, the authors provide examples and emphasize the impact of such attacks. Yampolskiy *et al.* [128] placed their emphasis on how cyber attacks might influence the physical space and proposed a taxonomy that could be used to categorize cross-domain attacks as well. This approach is advantageous for IoT systems, because many such systems have cyber-physical components and the cross-domain aspect of the taxonomy can capture affects on edge and cloud components as well. The proposed taxonomy has six dimensions, as follows:

- Influenced Element: The object that is directly manipulated by an attack.

[5] https://attack.mitre.org/, Last visited: Apr 30, 2021

- Influence: It describes how the Influenced Element is manipulated.
- Victim Element: A list of other elements that become manipulated via interactions.
- Impact on Victim: It describes what impact the attack has on the Victim Element.
- Attack Means: It defines the manipulation performed on the Influenced Element.
- Preconditions: It defines the conditions necessary for the attack to lead the results captured in Victim Element and Impact on Victim.

Unfortunately, none of the dimensions has categories to aggregate information. This lack of structure in the taxonomy makes it difficult to efficiently process attack information. In order to demonstrate this shortcoming, let us consider CVE-2015-1179:

```
Multiple cross-site scripting (XSS) vulnerabilities in
data_point_details.shtm in Mango Automation 2.4.0 and earlier allow
remote attackers to inject arbitrary web script or HTML via the (1) dpid,
(2) dpxid, or (3) pid parameter.
```

The Attack Means for this CVE can be formulated in multiple ways. It can be specified as "inject arbitrary web script or HTML", but "XSS" and "cross-site scripting" are also valid choices. The Precondition is that the attacker has to get the victim to follow the crafted link, which can be achieved in many different ways (e.g., via e-mail, messages, embedding in other websites). As a result, the value chosen for the Precondition dimension can be specified in multiple ways. The Influenced Element dimension can also hold multiple valid values, including `data_point_details.shtm`, Mango Automation 2.4.0, or the device running this software. Specifying the Victim Element is challenging in itself. Interacting with the Influenced Element via a browser definitely influences the browser processing the injected web script or HTML. However, depending on the specifics of the script, the interaction may also affect user data (e.g., by stealing credentials, session data, etc.) and/or the user (e.g., sending HTTP requests in the user's name). In summary, the flexibility provided by this taxonomy is advantageous for human interpretation, but poses challenges for automated processing and data aggregation.

In summary, existing taxonomies for embedded systems have one of the following two shortcomings. They are either limited to a single application domain [135, 38] or too generic [128] for automated data processing. To overcome these shortcomings, we extend the existing attack taxonomies, and modify their contents and structures.

2.2.2 Proposed attack taxonomy

Based on existing attack taxonomies and attacks, we 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 we 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. Taking a value from each category gives a 5-tuple that characterizes an attack scenario.

We populated the above dimensions by going through the 106 selected CVE records and observing the preconditions and the type of the vulnerabilities described in the CVE records, as well as the targets, methods, and effects of the potential attacks that may exploit the described vulnerabilities. As for the preconditions, we observed the following types of requirements for the attacker:

- **Internet facing device:** Many vulnerabilities in the CVE records are potentially exploitable by a remote attacker if the device is connected to the Internet. The attacker does not necessarily need to have access privileges; the only requirement is that the attacker can potentially discover the device and send messages to it via the network.
- **Local or remote access to the device:** This precondition requires the attacker to have some privileges that allow for logical access to the services or functions provided by the device. This logical access can be restricted to local access or it can be a remote access capability (e.g., via the Internet). Often, the privileges required by the access are normal user privileges, and not administrator privileges.
- **Direct physical access to the device:** Direct physical access requires the attacker to access the device physically. However, the attacker might not need any privileges to access the services of the device.
- **Physically proximity of the attacker:** In some cases, the attacker does not need physical access. It is sufficient that the attacker can be in the physical proximity of the device. For instance, attacks on wireless devices may only require being within the radio range of the target device.
- **Miscellaneous:** We observed a number of other preconditions in CVE records, each appearing in only one or a few records, and we decided to create this general category to represent them. For example, a miscellaneous precondition is when the target device has to run some software or has to be configured in a certain way for the vulnerability to be exploitable.
- **Unknown:** Some CVE records and other sources do not provide sufficient amount of information to determine the preconditions of a potential attack; in these cases, we classify the precondition as unknown.

The vulnerabilities that we observed in the CVE records have the following types:

- **Programming errors:** Many of the vulnerabilities in the selected CVE records stem from programming errors, which may lead to control flow attacks (e.g., input parsing vulnerabilities leading to buffer overflow problems, and memory management problems such as using pointers referring to memory locations that have been freed).
- **Web based vulnerability:** Many embedded devices have a web based management interface through which they can be configured and updated. However, the web server applications running on those devices are typically rarely updated. Hence, those devices are exposed to web based attacks that exploit unpatched vulnerabilities in the web-based interface of the device.
- **Weak access control or authentication:** Many devices use default or weak passwords, and some devices have hard-coded passwords that provide backdoor access to those who know the hard-coded password. Such vulnerabilities make it possible for attackers to bypass access control mechanisms rather easily with minimal effort.
- **Improper use of cryptography:** Some devices use cryptographic mechanisms for authentication purposes or for preserving the confidentiality of some sensitive information. Often, cryptographic mechanisms are not used appropriately, which leads to fatal security failures. Examples include the use of weak random number generators for generating cryptographic keys, or vulnerabilities in the protocols that use cryptographic primitives.
- **Unknown:** Similar to precondition, some CVE records do not contain information about the vulnerability itself, while they described the target and the effect of the potential attacks exploiting the unspecified vulnerability.

Regarding the target of the potential attacks, we distinguish the following layers of the embedded system architecture: **hardware**, **firmware/OS**, and **application**. When no specific layer can be determined from the CVE record, or when a potential attack can target multiple layers, we identify the **device** itself as the target of the attack. Note that we do not make a clear distinction between the firmware and the operating system (OS), because in many cases, embedded devices have no real OS, but their firmware provides typical OS functionality (such as control to resources). In addition, we observed that the attacks may not target the embedded devices themselves, but many CVE records report potential attacks on the **protocol** used by those devices for communication or device management.

The attacks that we observed in the selected CVE records cover a wide range of methods. We grouped them in the following types:

- **Control hijacking attacks:** This type of attacks divert the normal control flow of the programs running on the embedded device, which typically results in executing code injected by the attacker.

- **Reverse engineering:** Often, an attacker can obtain sensitive information (e.g., an access credential) by analyzing the software (firmware or application) in an embedded device. This process is called reverse engineering. By using reverse engineering techniques, the attacker can find vulnerabilities in the code (e.g., input parsing errors) that may be exploited by other attack methods.

- **Malware:** An attacker can try to infect an embedded device with a malicious software (malware). There are different types of malware. A common characteristic is that they all have unwanted, potentially harmful functionality that they add to the infected system. A malware that infects an embedded device may modify the behavior of the device, which may have consequences beyond the cyber domain. For instance, the infamous Stuxnet worm reprogrammed PLCs in an uranium enrichment facility, which ultimately led to the physical destruction of the uranium centrifuges controlled by the infected PLCs.

- **Injecting crafted packets or input:** We observed that injection of crafted packets is an attack method against protocols used by embedded devices. A similar type of attack is the manipulation of the input to a program running on an embedded device. Both packet and input crafting attacks exploit parsing vulnerabilities in protocol implementations or other programs. In addition, replaying previously observed packets or packet fragments can be considered as a special form of packet crafting, which can be an effective method to cause protocol failures.

- **Eavesdropping:** While packet crafting is an active attack, eavesdropping (or sniffing) is a passive attack method whereby an attacker only observes the messages sent and received by an embedded device. Those messages may contain sensitive information that is weakly protected or not protected at all by cryptographic means. In addition, eavesdropped information can be used in packet crafting attacks (e.g., in replay type of attacks).

- **Brute-force search attacks:** Weak cryptography and weak authentication methods can be broken by brute force search attacks. Those involve exhaustive key search attacks against cryptographic algorithms such as ciphers and MAC functions, and dictionary attacks against password based authentication schemes. In both cases, brute-force attacks are feasible only if the search space is sufficiently small. Unfortunately, we observed CVE records that report such vulnerabilities.

- **Normal use:** This refers to the attack that exploit an unprotected device or protocol through normal usage. This is because we observed CVEs that reported on potential attacks where the attacker simply used some unprotected mechanism as if he was a legitimate user. For instance, the attacker can access files on an embedded device just like any other user if the device does not have any access control mechanism implemented on it.

- **Unknown:** We observed some CVEs that described vulnerabilities but did not

identify any particular attack method that would exploit those identified vulnerabilities.

The effect of the above attacks can be the following:

- **Denial-of-Service:** Many CVE records identify potential attacks that lead to denial-of-service conditions such as malfunctioning or completely halting the device.
- **Code execution:** Another large part of the analyzed CVE records identify execution of attacker supplied code on the embedded device as the effect of potential attacks. This also includes web scripts and SQL injections, not only native code of the device.
- **Integrity violation:** A commonly observable effect of potential attacks is the integrity violation of some data or code on the device. This includes modification of files and configuration settings, as well as the illegitimate update of the firmware or some applications on the device.
- **Information leakage:** In some cases, the effect of the attack is the leakage of some information that should not be obtained by the attacker.
- **Illegitimate access:** Many attacks result in the attacker gaining illegitimate access to the device. This not only includes the cases when an attacker, who otherwise has no access to the device, manages to logically break into it, but also cases when the attacker has already some access, but he gains more privileges (i.e., privilege escalation).
- **Financial loss:** Certain attacks enable the attacker to cause financial loss to the victim, e.g., by making calls from a smart phone. Actually, most attacks can lead to financial loss in a general sense, so we use this criterion to represent only those attacks whose primary goal is to cause financial loss. A typical example would be an attack that aims at sending and SMS or making a call to a premium number from a compromised smart phone.
- **Degraded level of protection:** In some CVE records, we observed that the potential attack results in a lower level of protection than expected. For example, a device could be tricked into using weaker algorithms or security policies than those that it actually supports.
- **Miscellaneous:** Some attacks cause users to be redirected to malicious websites or traffic to be redirected. In these cases, there is not enough information about what happens exactly to the redirected user or traffic, but there is information about the effect.
- **Unknown:** In some CVE records, no specific attack effect is identified. This is mainly the case when the attack method is not identified either, and the CVE record contains only information about a vulnerability.

In the rest of this section, we will show how the taxonomy can be applied to CVE entries through two examples. Firstly, the description of CVE-2006-2560 is as follows:

```
Sitecom WL-153 router firmware before 1.38 allows remote attackers to
bypass access restrictions and conduct unauthorized operations via a
UPnP request with a modified InternalClient parameter, which is not
validated, as demonstrated by using AddPortMapping to forward arbitrary
traffic.
```

The description mentions a remote attacker without authentication, who has to be able to send packets to the **Internet facing** device. The description also states, that the exploitation is done via a modified parameter, which means that the attack method is **Injecting crafted packets or input**. The entry also tells us that the parameter is not validated by the **firmware**, so the vulnerability is a **programming error**. By exploiting this vulnerability, the attacker can conduct unauthorized operations, so the effect is **illegitimate access**.

As another example, let us consider CVE-2008-1262:

```
The administration panel on the Airspan WiMax ProST 4.1 antenna with
6.5.38.0 software does not verify authentication credentials, which
allows remote attackers to (1) upload malformed firmware or (2) bind
the antenna to a different WiMAX base station via unspecified requests
to forms under process_adv/.
```

The attacker remotely sends messages to the **Internet facing** device. He is able to upload a malformed **firmware**, which results in an **integrity violation**. The antenna does not verify authentication credentials and thus implements **weak access control or authentication**. As a result, the attacker does not need to do anything suspicious: he has to access the device, type in credentials when prompted and upload the modified firmware, which is a completely **normal usage**.

Finally, let us consider CVE-2015-1179:

```
Multiple cross-site scripting (XSS) vulnerabilities in
data_point_details.shtm in Mango Automation 2.4.0 and earlier allow
remote attackers to inject arbitrary web script or HTML via the (1) dpid,
(2) dpxid, or (3) pid parameter.
```

The CVE record describes a cross-site scripting vulnerability, which is a well-known vulnerability of **Web Applications**. Such vulnerabilities are easily exploited via HTTP requests sent to **Internet facing** devices. In this case, exploitation requires the attacker to **inject** malicious Javascript code into one of the specified parameters. Successful exploitation results in **code execution** in the victim's browser.

2.2.3 Evaluation of proposed attack taxonomy

To evaluate our taxonomy, we applied it to the subset of 3826 CVEs related to embedded systems. Applying the taxonomy was done in a half-automated and iterative way. We

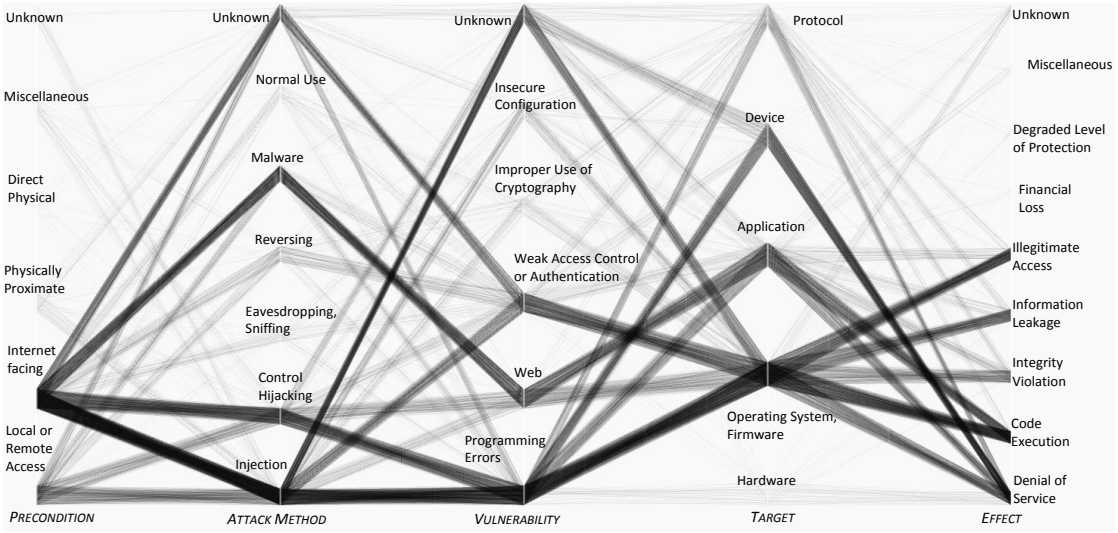

Figure 2.1: Common attack scenarios for embedded IoT devices

created a Python script[6] that used expressions of CVE entries related to a particular category in each dimension of our taxonomy. For example, obtaining some kind of access to the system is common in the Illegitimate Access category of the Effect dimension. When the script encountered an entry for which it could not determine the correct category, it displayed the description of the entry and the relevant expression had to be added manually. We repeated this procedure until all CVE entries were categorized.

The immediate result of our script shows that many CVE entries can be put into multiple categories. And this outcome is natural if we look at some examples. For example, CVE-2010-0597 tells us that the vulnerability "allows remote authenticated users to read or modify the device configuration, and gain privileges or cause a denial of service (device reload)". Reading the configuration discloses sensitive information to the attacker, writing the configuration violates the integrity of the configuration, privilege escalation is a type of illegitimate access and then there is denial of service, which is a single category in itself. It depends on the attacker what effect he wishes to have. In addition, if his actions had multiple effects, his attack could be categorized into multiple categories in the Effect dimension. This observation stands for the attack method as well since there are attacks that require multiple steps to be performed by the attacker. The description of CVE-2009-1477 states that certain switches have hard-coded SSL private keys, which allows the attacker to decrypt HTTPS sessions. Exploiting this vulnerability requires the attacker to obtain the hard-coded key from a previous installation, and then sniff the channel for messages to decrypt. Our taxonomy allows an attack to be classified into multiple categories, thus variations of attacks can be handled easily. However, to create statistics and for the sake of manageability, we needed to simplify the output of

[6]The created scripts can be found at http://www.hit.bme.hu/~buttyan/publications.html#PappMB15pst

our script. We chose the following approach: when a CVE entry could be categorized into multiple categories, we have manually selected the scenario we think has the most possibility of happening.

The output of our script was a table in which each row represents a vector from our taxonomy, a possible attack scenario. Vectors have five coordinates, each coordinate representing a category from our five dimensions. Figure 2.1 shows vectors on a parallel coordinates diagram. Each path on the diagram is an attack scenario described by the vector; the categories the path touches in each dimension show a different aspect of the attack. The thicker a path, the more CVE entries mention it as a possible scenario. It is clear from the figure that most attacks require only that the attacked device should have a public IP address. Local or remote access (some kind of user authentication) is also often needed to launch the attack.

The attacker has multiple methods available, but according to the CVE entries, most of the time he will either inject crafted inputs and arguments, or perform a control hijack by exploiting buffer overflows or embedding commands into parameters. Another common exploitation method is the use of malware, when the attacker injects scripts into web pages or is able to install a compromised firmware. Numerous CVE entries do not state how the vulnerability could be exploited.

CVE entries suggest that embedded systems have three common vulnerabilities: programming errors, web-based vulnerabilities and weak access control or authentication. Figure 2.1 also shows common ways to exploit the vulnerabilities. A programming error can be exploited by control hijacking and crafted inputs. A malicious script often exploits a web-based vulnerability. Weak access control or authentication is also exploitable by crafting inputs, e.g., directory traversal vulnerabilities. It must also be stated that a considerable amount of CVE entries do not disclose the vulnerability.

The possible target of the attack is usually (although sometimes indirectly) mentioned in the entries. Operating systems and firmware suffer the most attacks: programming errors in these pieces of software and weak access control or weak authentication enable attacks at the lowest software-based level but the undisclosed vulnerabilities often affect these pieces of software as well. Applications could be targeted via exploiting programming errors and web-based vulnerabilities. Many black lines touch the Device category because when the entry did not provide any information about which part of the system if affected by the attack, we classified the target into the Device category. Another observation here concerns protocols: this category is closely related to programming errors which tells us that most of the time the implementation of protocols contain exploitable vulnerabilities and it is not the design of the protocol that is flawed.

Embedded systems can be affected by attacks in multiple ways. Denial of service situations are quite commonly mentioned, especially if there is no accurate information on the target of the attack. The attacker may also be able to execute code, either his own or a program installed on the system. Operating systems and firmware suffer the most integrity violation: the attacker might compromise sensitive files or install a new firmware. Information leakage affects operating systems and firmware as well in situations when the attacker cannot modify an arbitrary file but is able to read said file or

when sensitive information, e.g., version information is disclosed. Naturally, illegitimate access is closely related to operating systems and firmware through privilege escalation and impersonation.

2.3 Conclusion

This chapter provided a comprehensive overview of the threat landscape of embedded IoT devices and systems by describing both attacks and vulnerabilities. Based on the discussed attacks and vulnerabilities, as well as existing taxonomies, we proposed a new attack taxonomy to classify and describe common attack scenarios against embedded IoT devices and systems. The attack taxonomy derived in this chapter provides information on how an embedded system can be attacked. Moreover, the structured knowledge can assist analysis and design of systems including or based on embedded devices during system development lifecycle. The results were published in [C1, C2].

In its present state, the attack taxonomy also helps us to forecast trends in embedded-system security. Given the recent trends in machine-to-machine communications and the growing number of embedded devices with Internet connection, we expect Internet facing devices to continue to suffer the majority of attacks. These attacks can come in many forms; however, based on the results of applying our attack taxonomy to CVE records, malware and exploits are the most common approaches.

Consequently, the security of embedded IoT devices could be improved significantly with better protection from malware. This observation is the key motivator behind our research presented in Chapters 3 and 4 in the fields of malware clustering and malware analysis, respectively. In order to deal with exploits, the targeted vulnerabilities could be uncovered more reliably and patched individually. However, this approach requires significant effort each time a vulnerability is uncovered, without the guarantee that all vulnerabilities have been dealt with. Therefore, in Chapter 5, we take a different approach and present a new mode of operation for embedded IoT devices. This mode of operation give the device the ability to periodically self-heal and cleanse itself of compromises.

In the future, the presented taxonomy may require adjustment to accurately reflect the changes in technology and attacks. This need for change can be detected by checking the number of entries in the Unknwon categories of each dimension. A growing number of entries there can signal the need to adjustment. Depending on the cause for too many CVE entries in the Unknown, it is either the processing script that needs adjustment or the categorizes in the taxonomy. The first scenario requires additional key-words to be added to the processing scripts. The second is a more fundamental change to the presented taxonomy, which affects not only automated processing but also requires an in-depth look at the landscape of the security of embedded IoT devices.

Chapter 3

Malware clustering using binary similarity hashes

The concept of malicious software – or malware, as it is called in the computer security community – is almost as old as computers themselves. While in early years, malware was mainly created for fun or for experimental purposes, with the growing number of personal computers and the proliferation of Internet connectivity, malware development became a profitable business for miscreants at the end of the last century. Later, smart phones appeared, and attackers started developing malware for mobile devices. Today, we are witnessing a new trend: all sorts of embedded devices are being connected to the Internet, which is rapidly transforming into an Internet of Things, or IoT for short. Not surprisingly, malware development followed this new trend, and malware is now developed for embedded IoT devices as well.

A significant problem is that the number of IoT devices is already large and grows exponentially, which means that they can be converted into a substantial attack infrastructure by infecting them with malware and organizing them into botnets. Such botnets have already appeared in the wild. An infamous example is the Mirai botnet, and the importance of the problem is illustrated by the fact that it holds the record for the most intensive DDoS attack in history ever [5]. Of course, malware infected IoT devices can be used not only for building botnets, but also for all sorts of other misdeeds, such as click fraud and bitcoin mining.

Anti-virus companies rely on malware classification methods 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.

In this chapter, we investigate the possibility of clustering malware samples based on their TLSH similarity score, where TLSH is the Trend Micro Similarity Hash, a binary similarity hash algorithm developed by Trend Micro [91]. While this approach can be used for clustering any malware, here we use it to cluster IoT malware samples due to the importance of this new trend. We study the performance of two distance-based clustering

algorithms, k-medoid and OPTICS, on a large corpus of IoT malware samples when they are used with the TLSH similarity metric to measure distances between samples. Our results show that neither of the two algorithms have acceptable performance: k-medoid produces clusters with unacceptably large diameters, meaning that it puts unrelated samples into the same cluster, whereas OPTICS fails to cluster more than half of the samples in our data set. To overcome these problems, we propose a new clustering algorithm, which is based on OPTICS and achieves a performance superior to both k-medoid and OPTICS.

The organization of the chapter is as follows. Section 3.1, discusses program similarity measures. Section 3.2 gives an overview of our research methodology, including how we obtained our initial set of IoT malware samples, how we cleaned this initial corpus, and how we determined the TLSH difference threshold under which two samples are considered variants of the same malware. Section 3.3 presents the performance of the k-medoid and OPTICS clustering algorithms on our corpus and explain why they are not appropriate for malware clustering. We describe our own clustering algorithm in Section 3.4, evaluate its performance, and compare it to that of k-medoid and OPTICS. We present other approaches to clustering malware in Section 3.6. Finally, we conclude this chapter and sketch some possible future work in Section 3.7.

3.1 Background on binary similarity hashes

Binary similarity hash algorithms are a popular group of static analysis techniques, which can compare programs and find similar instances. The input of such algorithms is the raw sequence of bytes and their output is a specially constructed fingerprint or digest, also called the similarity hash value. The similarity hash value is constructed in such a way that slight changes to the input sequence of bytes results in only a slight change to the resulting similarity hash value. Binary similarity schemes also have comparison algorithms unique to each scheme that take two similarity hash values as input and output a difference score. Different schemes interpret scores differently; for example, higher scores for ssdeep [68] and sdhash [106] signals a higher degree of similarity, while in the case of TLSH [91], the lower the score, the more similar the inputs are.

Currently, ssdeep and sdhash are treated as the industry best practice. Many companies, including VirusTotal, VirusShare and Malwr list the similarity hash of samples for these algorithms. ssdeep generates string hashes up to 80 bytes that are concatenations of 6-bit piece-wise hashes. The hash value then can be compared with other hashes to measure of how many character operations are necessary to transform one string into the other. Because of the fixed-size hash it produces, it quickly loses granularity and only works for relatively small files of similar sizes. sdhash, on the other hand, is slower than ssdeep in terms of computation performance but it also overcomes ssdeep's main weakness: sensitivity to byte ordering. The scoring method used by sdhash, however, results in the undesirable property that similarities are not symmetric, i.e., $sim(A, B) \neq sim(B, A)$.

Recent research [92] have shown that TLSH is not only more precise than previous

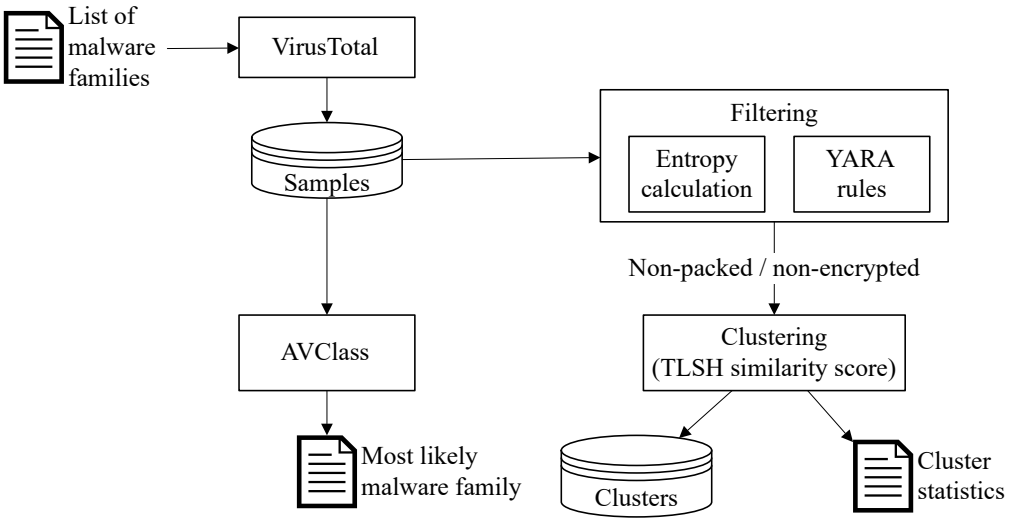

Figure 3.1: Overview of the methodology for evaluating clustering algorithms

methods, including ssdeep and sdhash, it is also applicable for malware classification. The TLSH digest of an input byte string, e.g., malware sample, is calculated in four steps. First, the byte string is processed in a 5-byte-long sliding window and the different byte triplets in the input are counted. In the case of binary executables, the sliding window captures the correlation between neighboring instructions, which encode the executable's functionality. Second, so called quartile points are calculated which separate the counter values from the previous step into four equal regions. In the third step, the 3-byte-long digest header is constructed. The first byte of the header is a checksum of the byte string, the second byte represents the logarithm of the byte string (modulo 256), and the third byte is derived from the quartile points. The remainder of the digest is computed based on the counter values. The result of these steps is a 70-byte-long digest.

TLSH, similarly to other static analysis techniques and binary similarity hashes, performs well as long as the input binary is not packed or encrypted. Packed executables contain only a small portion of executable code; most of the files' contents are filled with high entropy data. High entropy data decreases the accuracy of calculated TLSH differences, resulting in detecting samples of the same variants as dissimilar.

3.2 Methodology

This research is concerned with identifying groups of similar malware using their TLSH similarity score. The high-level overview of the methodology we followed during this research is shown in Figure 3.1. The methodology can be divided into three main steps:

1. data collection, which results in a data set of IoT malware samples,
2. filtering, which removes packed and/or encrypted samples from the data set, and

3. clustering, which identifies variants in the data set by grouping malware samples based on their pair-wise TLSH differences.

To evaluate cluster configurations, we need the TLSH difference threshold denoting variants of malware families.

3.2.1 Data Collection

In order to acquire a data set of IoT malware samples, we first need to select an IoT-relevant embedded architecture, which the data set should target. This is a necessary step, as different instruction sets could cause TLSH to measure big differences between variants compiled for different architectures. For this study, we select samples targeting the ARM architecture due to its widespread use in the IoT world. Secondly, we compile a list of 29 malware family names based on previous studies of the IoT malware landscape [36, 32, 122]. These malware families specifically target the IoT ecosystem. Many of them implement the ability to infect other machines and connect them to an existing botnet. The attacker remotely administers the botnet via various channels, e.g., IRC or HTTP-based communication. Samples from these families take various commands from the attacker via a command & control server. The families also share similar traits as they are known to copy and develop features from each other, e.g., after Mirai's source code was leaked[1], several modifications led to the branches Satori, Okiru, Masuta and PureMasuta.

We use the compiled list to search for and download malware samples from VirusTotal [2], a publicly available site to which users can upload executables and submit URLs. The site scans uploaded executables with a number of anti-virus tools and returns to the user the collected results, including the malware family names assigned by anti-virus tools. We downloaded 12,993 samples and their corresponding anti-virus scan results.

The scan results from VirusTotal are fed to AVClass [110], which outputs the most likely malware family name based on a majority vote cast of anti-virus tools' assigned labels. We made changes to the tool's source code because initially, it could not provide a label for a number of samples. In order for AVClass to cast a majority vote, it needs at least 4 detections per sample. As some of our samples have lower detection rates, we remove this requirement. Throughout our study, we use AVClass's output as the ground truth for all samples.

3.2.2 Filtering

The second step in our methodology is to filter the downloaded samples. The step is required because TLSH has difficulty identifying similarities between packed and/or encrypted samples. We use two approaches for filtering our data set. Firstly, we use binary entropy calculation, which calculates the empirical entropy of a file based on the

[1] https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/ (Last visited: Dec 30, 2020)

[2] https://www.virustotal.com/ (Last visited: Dec 30, 2020)

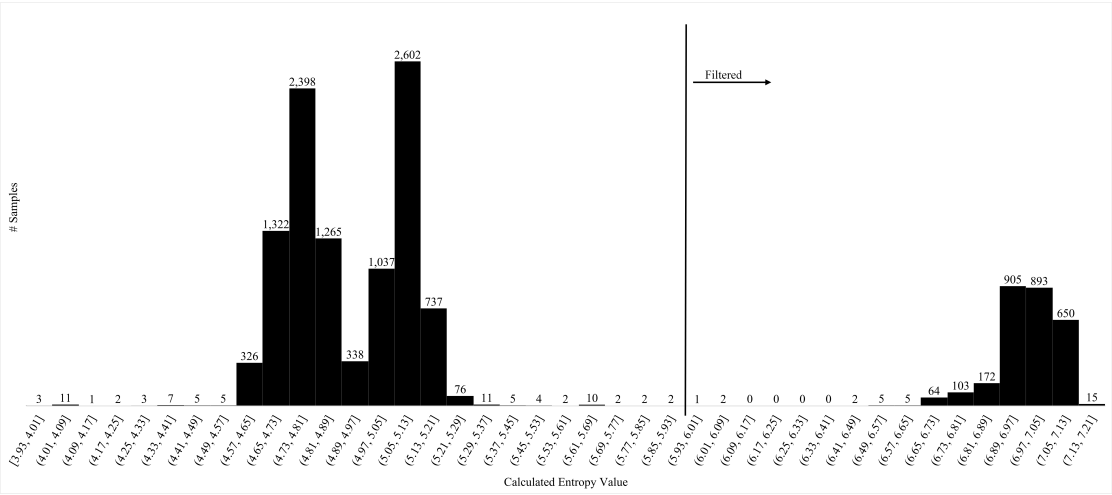

Figure 3.2: Entropy distribution of the data set used to evaluate clustering algorithms

contained bytes. There exist best practices for calculated entropy values signaling packed and/or encrypted executables [75]. The measured empirical entropy values of our data set is shown in Figure 3.2. There is a clear cut between the set of native executables and the set of packed and/or encrypted samples. As a result, we excluded 2,817 samples.

Table 3.1: Malware families in the filtered corpus for evaluating clustering algorithms

Family	# Samples	Family	# Samples
mirai	6,108	ditertag	2
gafgyt	3,711	oneeva	2
dofloo	163	cloxer	1
tsunami	92	zergrush	1
ddostf	63	luabot	1
presenoker	19	lightaidra	1
dnsamp	5	no name recovered	7

Binary entropy calculation has one major limitation, namely, that large sections of low-entropy bytes can lower the calculated overall entropy. In order to overcome this limitation, we also use YARA-rules[3], as packers can leave traces in the binary, e.g., specific strings and/or byte sequences unique to the packer. We run YARA-rules for UPX and other packers on our whole data set, looking for packed binaries. We found a total of 980 packed samples, all of which were packed with UPX and have already been filtered using binary entropy calculation. Table 3.1 shows the distribution of malware families in our filtered data set.

[3] https://yara.readthedocs.io/en/latest/ (Last visited: Dec 30, 2020)

3.2.3 TLSH difference threshold selection

Before being able to cluster our data set, we needed a TLSH difference threshold signaling variants of malware families that produce zero false matches. This work was carried out by Tamás [117], however, we summarize his work here for completeness. In [117], he suggested a threshold of 70 to be used for few false positives. This suggestion, however, was based on a relatively small set of 477 samples with most of the samples belonging to two families. In order to define a globally applicable TLSH difference threshold for malware similarity, he carried out a measurement at a much bigger scale.

He used the EMBER data set [3], the largest available labeled malware data set at the time of this research[4]. He processed its test set, containing 100,000 malicious samples from 917 malware families. As the actual malware binaries required to calculate the TLSH digests are not included in the data set, his measurements were carried out on the 62,863 samples available in Ukatemi Technologies's malware repository. In order to find the maximal zero false positive threshold, he searched the available sample set with a candidate threshold. If false matches are found, the threshold is reduced. This process yielded the zero false positive threshold of 1, which is a much lower threshold than we anticipated. Manual investigation based on VirusTotal details and behavior pages, as well as IDA Pro[5] with the Diaphora plugin[6] revealed that the EMBER data set labels are often incorrect. Samples with very similar functionality and behavior are labeled as members of different malware families.

Therefore, he resorted to manual verification. He randomly select samples from a data set of 355,795,714 unlabeled malware samples, courtesy of Ukatemi Technologies. The selected samples include 9 Azorult samples, 3 Lightneuron samples and 10 Pioneer samples. He compared these samples to the rest of the data set using the above-mentioned methodology. His analysis yielded the TLSH difference threshold of 48.

3.2.4 Clustering

The final step is clustering the data set. Our goal is to group samples based on their TLSH difference, thereby detecting variants of malware families. Initially, we clustered the data set with two widespread algorithms, k-medoid and OPTICS, using TLSH difference as the distance metric. The results, presented in more details in Section 3.3, show that k-medoid often puts unrelated data into the same clusters, while OPTICS fails to cluster more than half of the samples in our data set. Therefore, we develop a new clustering algorithm, which we present in Section 3.4.

3.3 Performance of existing clustering algorithms

We now elaborate on the performance results of clustering our data set using both k-medoid [60] and OPTICS [4]. k-medoid is a PAM-based algorithm in which clusters can

[4] 2018, results were published in [C3].
[5] https://www.hex-rays.com/products/ida/ (Last visited: Dec 30, 2020)
[6] https://github.com/joxeankoret/diaphora (Last visited: Dec 30, 2020)

Table 3.2: 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

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.

OPTICS 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 ϵ values for data points such that data points have at least $minPts-1$ samples in their ϵ radius. OPTICS also has a built-in clustering algorithm, ξ, which clusters data points by detecting abrupt changes in the ϵ-values.

3.3.1 k-medoid

There are several rationales behind choosing k-medoid as the clustering algorithm. It is unsupervised, i.e., 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 in our scenario, because cluster heads can represent other malware samples in the same cluster. The disadvantage of this algorithms is that the input k has to be specified.

Unfortunately, we do not know how many variants there are in our data set, therefore, we calculate cluster configurations for all potential k values. We compute the $s(k)$ metric for all cluster configurations in order to rank our setups.

As shown in Table 3.2, 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. We 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, cluster heads of different clusters should have a TLSH difference score higher than 48. With this requirement in mind, we looked at our 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 3.3. In this configu-

Table 3.3: 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

ration, the calculated cluster diameters range from 180 to 1,038, the mean being 468. In our scenario, cluster diameter is interpreted as the largest TLSH difference between any two samples in the same cluster. Taking our TLSH difference threshold for variants of 48 into consideration, we can conclude that clusters in this setup contain very different samples; clusters should be split into smaller clusters.

Cluster sizes in the setup range from 35 to 1,110 samples. However, as shown in Table 3.1, certain families contribute only a small number of samples to our data set. The configuration does not reflect that distribution, as it does not have any singleton or small clusters. Thus, we conclude, that the k-medoid algorithm does not perform well for the goal of clustering malware samples based on TLSH differences.

3.3.2 OPTICS

The second algorithm we tested was OPTICS [4], a density-based algorithm. It is able to detect and cluster dense and sparse regions in the data set. This characteristic makes it favorable in our scenario as we have families with only a few samples as well as families with thousands of samples, as shown in Table 3.1.

The algorithm takes two additional parameters besides a precomputed distance matrix: $minPts$ and ϵ_{max}. We can specify an upper bound for ϵ_{max} as the maximum TLSH difference in our data set. However, selecting $minPts$ is a challenge without knowledge about the internal structure of our data set. To gain this knowledge, we 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 are 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 we 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 our data set. Such a high number of unclassified samples is disadvantageous in our scenario, as many samples would require additional analysis.

3.4 New clustering algorithm

The performance of both k-medoid and OPTICS present issues for malware analysis. Firstly, k-medoid produces clusters whose diameters are too large to represent variants of malware families. We determined the threshold TLSH difference of 48 for variants, however, k-medoid's diameters range between 180 and 1,038. OPTICS's cluster diameters are more in line with our threshold value, however, as much as 68% of our samples are detected as outliers.

These algorithms were originally developed to cluster measurements that may be noisy. In order to remove noise, the data set must be cleaned and it must also be balanced. A balanced data set in our scenario requires exclusion of samples from families with very high and very low sample counts. Such a step, however, is undesirable as potential outliers may represent previously unseen variants or entirely new families.

In light of these challenges, we develop a new clustering algorithm that meets the following requirements. 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.

Our algorithm is based on OPTICS, however, we replace OPTICS's default clustering algorithm, ξ. Our algorithm can be divided into three major phases. In the first phase, we extract information about the structure of the data set using OPTICS. In the second phase, we generate a greedy, initial cluster configuration based on TLSH differences. In the last phase, we merge clusters in order to compensate for the greedy mechanism in the previous phase.

In order to extract structural information from the data set, we reuse OPTICS with input parameters $minPts$ and ϵ_{max}. OPTICS can compute the minimum ϵ values required to form a cluster around individual samples and the algorithm outputs this information in the form of a list. Elements of this list are s_i-ϵ_i pairs, where s_i is an individual sample and ϵ_i is the reachability value, i.e., the minimum radius of the area around s_i in which there are $minPts$ number of other samples. Samples with low ϵ_i values represent dense regions while samples with high ϵ_i values represent sparse regions. We sort the resulting ϵ_i values such that samples in dense regions come first in the list.

Our initial clustering, described by Algorithm 1, begins with no clusters and chooses the first sample with the lowest ϵ_i value. This sample represents the densest region of the data set and it becomes the first cluster head. We then put those samples into the selected head's cluster that are considered similar enough, captured by a threshold parameter. Subsequent cluster heads are selected iteratively based on the samples' ϵ_i value. Candidates with lower ϵ_i values are tried first and are selected as a next cluster head, if their dissimilarity to all existing cluster heads exceeds a predefined threshold value (see line 6). There may be cases when new cluster heads cannot be selected this way. In such cases, we disregard the pre-defined threshold value and select the sample that is the most dissimilar from the cluster head they are most similar to (see line 17).

Algorithm 1 Initial clustering algorithm based on OPTICS

Input:

$s_0, \ldots, s_n$: samples to cluster

$\epsilon_0, \ldots, \epsilon_n$: reachability values of samples $s_0, \ldots, s_n$ (computed by OPTICS)

$hThr$: TLSH similarity score threshold to keep cluster heads dissimilar

$cThr$: TLSH similarity score threshold for forming clusters

Output:

clusters: list of clusters. Each cluster is a list of samples

1: **begin**
2: $remaining \leftarrow s_0, \ldots, s_n$
3: $clHeads \leftarrow []$
4: $clusters \leftarrow []$
5: **while** *remaining* is not empty **do**
6: **if** $\exists s_i : (\forall j : \epsilon_i \leq \epsilon_j) \wedge (\neg \exists s_j \in clHeads : sim_{TLSH}(s_j, s_i) < hThr)$ **then**
7: ▷ s_i is a new cluster head
8: $append(clHeads, s_i)$
9: ▷ assign all similar enough samples to s_i's cluster
10: $cl_i \leftarrow \{\forall s_k \in remaining : sim_{TLSH}(s_k, s_i) < cThr\} \cup \{s_i\}$
11: **for all** s_k in cl_i **do**
12: $remove(remaining, s_k)$
13: **end for**
14: $append(clusters, cl_i)$
15: **else**
16: ▷ maximize dissimilarity from most similar cluster head
17: $nextHead \leftarrow s_i : \forall s_j \in remaining : max\{\forall s_l \in remaining : min\{\forall s_{cl} \in clHeads : sim_{TLSH}(s_{cl}, s_l)\}\}$
18: $append(clHeads, nextHead)$
19: ▷ assign all similar enough samples to s_i's cluster
20: $cl_{nextHead} \leftarrow \{\forall s_k : sim_{TLSH}(s_k, s_i) < cThr\} \cup \{nextHead\}$
21: **for all** s_k in $cl_{nextHead}$ **do**
22: $remove(remaining, s_k)$
23: **end for**
24: $append(clusters, cl_{nextHead})$
25: **end if**
26: **end while**
27: **end**

We repeat this process until all samples are clustered.

Algorithm 2 Merging initial clusters

Input:
clusters: list of clusters
threshold: TLSH similarity score threshold to merge clusters
maxInc: maximum increase allowed in cluster diameter when merging two clusters
Output:
$clusters_{final}$: modified list of clusters

1: **begin**
2: $clusters_{final} \leftarrow clusters$
3: **for all** (cl_i, cl_j) pairs of *clusters* **do**
4: $cl_{temp} \leftarrow merge(c_i, c_j)$
5: **if** $d(cl_{temp}) < max(threshold, maxInc * max(d(cl_i), d(cl_j)))$ **then**
6: $remove(clusters_{final}, cl_i)$
7: $remove(clusters_{final}, cl_j)$
8: $append(clusters_{final}, cl_{temp})$
9: **end if**
10: **end for**
11: **end**

The data set may contain dense regions whose diameter is larger than the maximum allowed dissimilarity. In such cases, the initial clustering strategy faces a limitation as it groups the center of the region into one large cluster and generates several small clusters on its perimeter. In order to overcome this challenge, we try to detect such perimeter clusters and merge them with the center cluster. We combine two clusters if the combined cluster's diameter either remains under a given threshold value, or merging increases the diameter of the center cluster by a fixed parameter. This strategy is shown by Algorithm 2.

3.5 Evaluation of proposed clustering algorithm

In order to evaluate the efficiency of our proposed clustering algorithm, we compared it against the results of both k-medoid and OPTICS. During evaluation, we took into consideration cluster diameters, the number of singleton clusters generated and two new measures of goodness. We used the following configuration for our clustering approach: $hThr = 150$ and $cThr = 20$ for Algorithm 1, and for Algorithm 2, $threshold = 48$ and $maxInc = 1.1$.

The number of generated clusters and singleton clusters are shown in Table 3.4. k-medoid and OPTICS both generate considerably fewer clusters than our algorithm does, more in line with the number of malware families our data set contains. Our algorithm generates 745 clusters of which 353 are singletons, a negligible amount compared to

Table 3.4: Comparison of the clustering methods

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

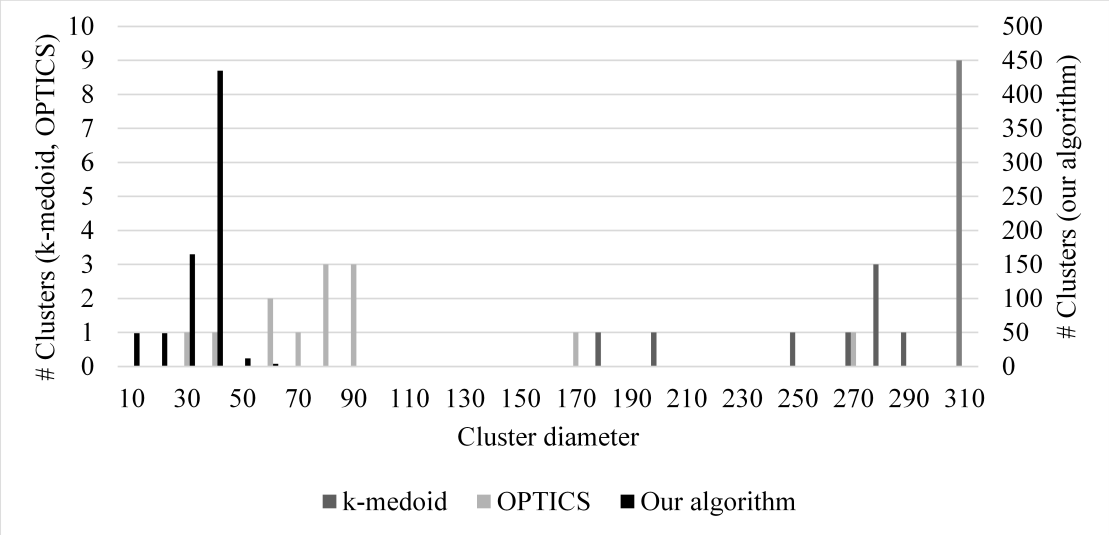

Figure 3.3: Cluster diameters produced by different clustering algorithms

OPTICS's performance.

The diameters of cluster configurations from all three algorithms are shown in Figure 3.3. Because our algorithm produced much more clusters than k-medoid and OPTICS, we use a different scale for the number of clusters in its case. Our experiments 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 our algorithm, however, is much closer to this threshold with 93.69% of our clusters having diameters below 50. As a result, our clusters are more likely to represent malware variants.

The first measure of goodness we 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 3.4 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 our algorithm, 185 have ratios over 0.6 of which 103 outperform both OPTICS and k-medoid.

We need to take into consideration that malware families share and/or copy features from each other. In response, the relaxation of our measure of goodness considers not

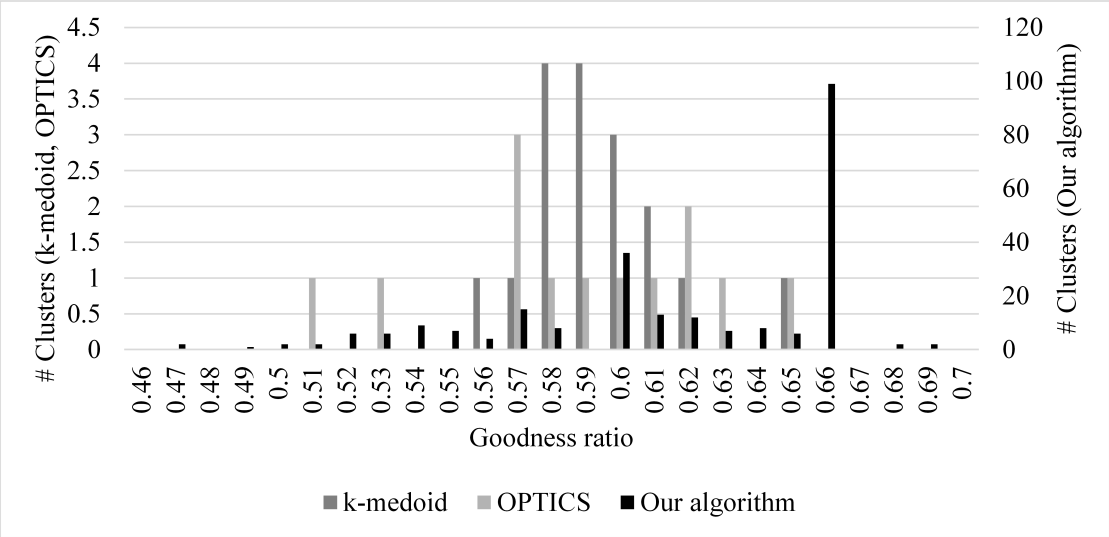

Figure 3.4: Goodness ratios produced by different clustering algorithms

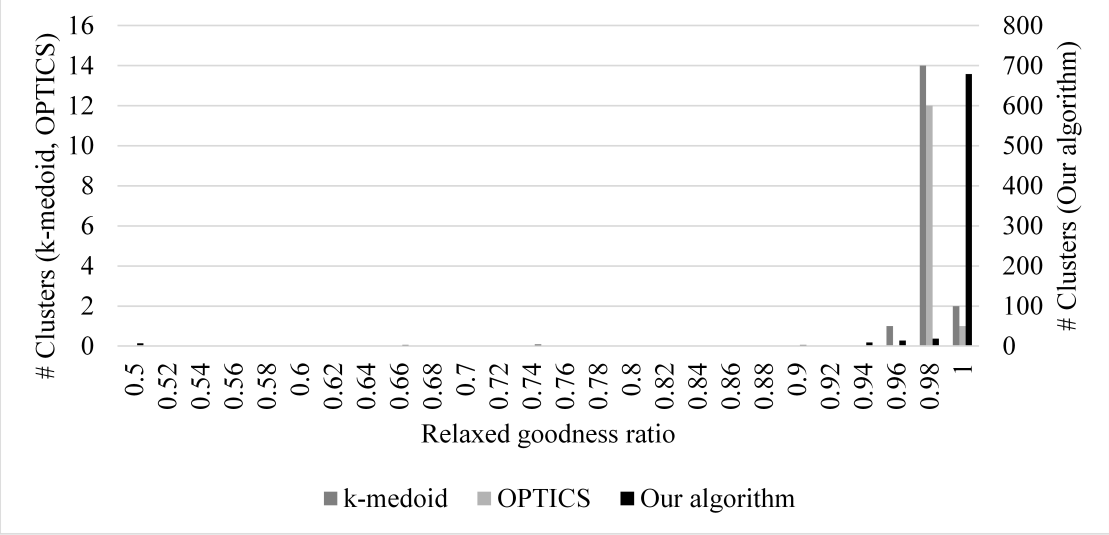

Figure 3.5: Relaxed goodness ratios produced by different clustering algorithms

only the family with the most samples in a given cluster but also families with which it is known to share features. For our data set, there exist known shared features among the Gafgyt/BASHLITE, the Kaiten/Tsunami, the Darlloz/Zollard, the Aidra/LightAidra, and the Mirai families. For this measure of goodness, singleton clusters can be included, because non-singleton clusters have a chance of achieving the ratio 1 thanks to the relationships between malware families. The figure shows that even though both OPTICS and k-medoid achieve ratios above 0.95, our algorithm outperforms both with almost all clusters achieving the measure of 1. However, our algorithm produces clusters whose relaxed ratios are well below those achieved by k-medoid and OPTICS. While investigating this issue, we found an indication of poor anti-virus labels. Specifically, all clusters achieving ratios of 0.5 contain 2 samples and their malware family labels do not match. However, the diameters of these clusters is quite low, the mean diameter being 30.71. Checking the samples' VirusTotal pages, we also saw that there were only a few labels on their scan pages. Therefore, the low ratios could be caused by misclassifications on AVClass's part.

3.6 Related work

Accurately identifying groups of similar malware in a large data set is an important problem for anti-virus companies. With rising tide of malware, these companies receive thousands of samples each day from various sources, including their intelligence networks, malware sharing sites, and specialized malware feeds. Manually dissecting and analyzing each sample does not scale well, therefore, there is a growing need to automate the process. The goal is to help analysts focus on those samples that are not similar to known malware. These samples are highly interesting for anti-virus companies because they could be indicators of highly sophisticated targeted attacks or new types of malware.

Due to the high interest in malware clustering, the field has a vast literature. Several surveys [130, 118, 45] have been published to organize existing research and systemize the lessons learned. In light of this, we only provide a short summary of the field of malware clustering here. Interested readers can look to the previously mentioned surveys for a more in-depth discussion.

The process of clustering malware samples typically starts with feature extraction. The goal of this step is to extract the main characteristics of individual samples and later use these characteristics to identify groups. Features proposed in literature can be categorized into static, dynamic and network features depending on the program analysis techniques used to extract them. Static features, such as byte sequences [131, 80], opcodes [52, 70, 120] and call graphs [53, 64, 67], are extracted using static program analysis techniques. TLSH similarity hashes can also be categorized as static features because their computation involves reading through the bytes in a sample and computing statistics over said bytes. Dynamic features are extracted using dynamic program analysis techniques and require samples to be executed in controlled environments, e.g., sandboxes. Examples of dynamic features include traces of system calls [103, 94, 95, 101] and behavioral profiles [11, 12, 81]. Network features [99, 115, 102] also require samples

to be executed, however, instead of focusing on events in the controlled environment, these features capture the behavior of a sample from the network's perspective.

Studies in literature typically explore a few types of features at most. However, in an industrial setting, features can transformed into vectors, in which individual coordinates correspond to different features. The resulting vector usually has high dimensionality. Therefore, dimensionality reduction and feature selection are important steps in the process [55, 119].

Existing research has explored a number of clustering algorithms as candidates for clustering malware samples. Most notable of these algorithms are hierarchical algorithms [98], PAM-based algorithms [83], k-nn [85] and DBSCAN [64]. Recently, there is an emerging trend of replacing clustering algorithms with machine learning algorithms [129], such as support vector machines [93], neural networks [66], and decision trees [84]. The expected benefit of this change is to automate feature selection and use machine learning algorithms to provide not only labels for individual groups for models for different malware families and variants.

3.7 Conclusion

In this chapter, we studied the applicability of TLSH in the field of malware clustering. We applied this approach to cluster IoT malware samples, however, we note that our approach is generic and can be applied to other malware as well. We studied the performance of two distance-based clustering algorithms, k-medoid and OPTICS, on a large corpus of IoT malware samples when they are used with the TLSH difference metric to measure distances between samples. We found that neither of the two algorithms had acceptable performance: k-medoid produced clusters with unacceptably large diameter, meaning that it put unrelated samples into the same cluster, whereas OPTICS failed to cluster more than half of the samples in our corpus.

To overcome these problems, we proposed a new clustering algorithm based on OPTICS, which achieved a better clustering performance. Our algorithm identifies dense regions in the data set and considers the data points in the center of the dense regions as cluster heads. A data point is included in a given cluster if and only if its TLSH difference to the cluster head is below a threshold value. We also merge clusters if they are close to each other in terms of TLSH difference and the diameter of the resulting merged cluster does not exceed a pre-specified threshold. We used the TLSH difference threshold below which two samples are considered variants of the same malware by empirical analysis of an independent set of malware samples (the EMBER data set). These results were published in [C3].

Chapter 4

Uncovering environmental requirements of malware

As mentioned in Chapter 3, 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. While not all instances of trigger-based behavior are malicious (take, for example, software easter eggs[1]), such behavior is often exhibited by malware. For example, malware can evade in-depth analysis by scanning its environment and ceasing malicious activities if it finds signs of an analysis framework[2]. Trigger-based behavior also includes backdoors, a behavior prevalent in embedded firmware images [31], in which case access is granted, if a specific string is received as input.

Due to the often malicious intent behind the implementation of trigger-based behavior, its detection is important. However, the specific conditions required to trigger the hidden behavior is known only to its author, therefore, uncovering such behavior via testing is challenging. Previous works in this field [16, 43, 65] have demonstrated the usefulness of symbolic execution [9] 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

[1] https://electrek.co/2017/12/23/tesla-christmas-easter-egg/ (Last visited: Jun 8, 2020

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

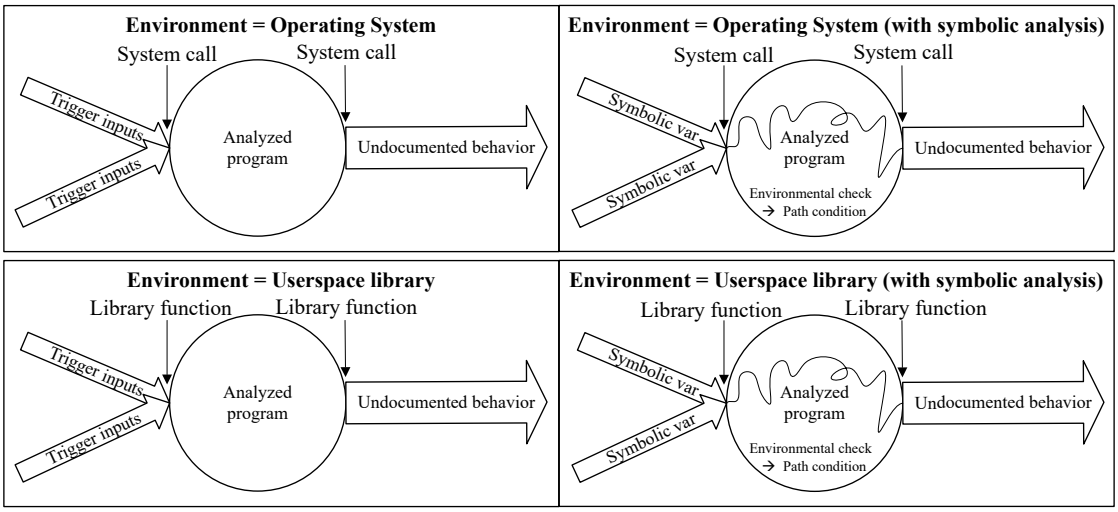

Figure 4.1: Using symbolic execution for recovering environmental conditions guarding malicious behavior

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 [43, 16].

In this chapter, we improve 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, our approach focuses on specific instances of malicious behavior. Malicious behavior can be modeled using library or system calls [19, 37, 20, 133, 71], depending on how the analyzed program's environment is defined. The overview of the main idea is shown in Fig. 4.1. We assume that the analyzed program is deterministic and interacts with the environment through 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. Our contributions are the following:

1. We present an approach for uncovering trigger-based behavior at both the source code and binary levels, which is capable of considering all external data sources as trigger input types. Our approach replaces library and system calls with symbolic summary functions, which return fresh symbolic variables instead of external data.

2. We validate the applicability of the general idea at the source code level using real-

world, open-source software samples. The results suggest that if symbolic execution succeeds, environmental conditions guarding the hidden, malicious behavior can be extracted from programs.

3. Our experiments at the source code level also show that our analysis is prone to the path explosion problem. In order to overcome this challenge, we develop a new path selection strategy relying on directed symbolic execution [77] to guide analysis towards selected program points.

4. Directed symbolic execution expects a semantically correct and complete interprocedural control-flow graph. The generation of such a control-flow graph is a challenge at the binary level, mainly due to indirect jumps. Therefore, we design our approach such that directed symbolic execution can be performed even if the interprocedural control-flow graph has incorrect/missing edges and/or nodes. We also discuss a technique to proactively check and add missing return edges to the graph throughout analysis, resulting in a more accurate view of the interprocedural control-flow graph.

5. We implement our approach in angr [111] for the binary level: we model 41 system calls for Linux and discuss modifications to angr's workflow in order to make our approach feasible in practice.

6. We evaluate our approach on an artificial malware sample with multiple instances of trigger-based behavior, as well as on a real malware sample compiled for the ARM platform. The program logics of the selected samples contain elements known to be challenging for symbolic execution and their execution relies on multiple sources of environmental input. Our approach is able to reach the specified program points and obtain the environmental conditions required to trigger their execution. In addition, our analysis has a reasonable performance considering the complexity of the analyzed samples and the generality of our approach.

The chapter is structured as follows. Section 4.1 provides an overview of symbolic execution and Section 4.2 discusses related work on triggering specific behaviors. Section 4.3 presents preliminary results on the validity of our approach when applied to source code. Section 4.4 builds on the results and presents an approach to uncover environmental conditions without *a priori* assumptions about trigger condition types at the binary level. Section 4.8 concludes the chapter and outlines future research directions.

4.1 Background on symbolic execution

The concept of symbolic execution [9, 109] has been well-researched over the years and as such, a full survey of the field is out of scope for this chapter. We only summarize its main characteristics and discuss the challenges it poses for our research.

Symbolic execution is an analysis technique originally proposed to automatically generate test cases and increase code coverage. During symbolic analysis, registers and

memory addresses do not store exact values but instead special symbols called symbolic variables. When first introduced into the analysis, symbolic variables may take on any value, i.e., they are *unconstrained*. When analysis reaches a branch in the analyzed program, two execution paths are spawned for both sides of the branch, i.e., it *forks*. In each spawned execution path, constraints are placed on the symbolic variables to represent the chosen path. The set of constraints collected on an execution path is the *path condition*. An execution path is *satisfiable*, if there exists an assignment to its symbolic variables such that the path condition is satisfied. If no such assignment exists, the execution path is said to be *unsatisfiable*.

The challenges of performing symbolic execution on arbitrary programs in binary form are manifold. First, tools implementing the technique need to model the execution state on the platform the analyzed program is intended to run on, including instruction set, registers, memory, calling conventions, flags, etc. Tools implementing symbolic execution, e.g., DART [47], KLEE [18], S2E [27], Mayhem [24] and angr [111], have such a model of the execution platform.

Second, symbolic execution can only reason about code it analyzes and has no inherent knowledge about library functions, system calls and their side effects. This is known as the *environment problem* and is typically tackled using summary functions [46, 2], which are pieces of code that summarize the effects of the missing piece of code.

Third, symbolic execution spawns execution paths to pursue at each encountered branch, resulting in an exponential growth in the number of execution paths with respect to the number of conditional branches. This challenge is known as the *path explosion problem* and as a result, symbolic execution cannot exhaustively explore all execution paths in all but the simplest of cases. This challenge is partially tackled by specifying which parts of the program are of interest to the analysis and only executing those parts symbolically. In such scenarios, the analysis engine keeps track of not only the symbolic state, but the concrete execution state as well, earning the name *mixed concrete and symbolic execution*. Previously mentioned tools implement such a mixed variant.

Lastly, since not all execution paths can be explored during symbolic execution, analysis has to prioritize. This challenge is known as the *path selection problem* and it is usually tackled using a heuristic exploration strategy. The depth-first strategy explores an execution path to completion before backtracking to the last fork and continuing with the second deepest branch. The breadth-first strategy, on the other hand, seeks to explore all execution paths in parallel. There are also randomized strategies, where the next pursued path is selected randomly with some probability. In certain application domains of symbolic execution, path selection algorithms have been tailored for a specific goal, e.g., maximizing coverage [18, 73] or reaching a certain program point [77, 96].

At the source code level, we use KLEE to validate the applicability our approach. At the binary level, we use angr, which has a model for the ARM platform. However, neither tools solve the environment problem and angr does not have built-in path selection strategies; our solutions are discussed in Sections 4.3 and 4.4.

4.2 Related work

We now present the existing literature related to our work. Uncovering hidden behaviors in malware is an important step in malware analysis as it allows for a more comprehensive view of analyzed programs. Such analysis helps detecting if a program executes differently in a controlled sandbox than in a real machine. This allows malware analysts to better understand programs and develop more accurate signatures for detection. Our work is closely related to this field, therefore, existing work is summarized in Section 4.2.1.

Section 4.4 introduces a path selection strategy for tackling the path explosion problem at the binary level. The building blocks for our proposed strategy have also been used to reproduce field failures in the software engineering field. Therefore, we provide an overview of this field in Section 4.2.2.

4.2.1 Uncovering hidden behavior in malware

There are three main approaches to uncovering hidden behaviors in the context of malware analysis. The main goal of all these approaches is to explore multiple execution paths in the analyzed program. Depending on the underlying program analysis techniques, the approaches must make a decision regarding a trade-off between precision and scalability.

The most scalable approach is known as *forced execution*. This technique relies on instruction-level monitoring to detect branches during execution. At branches, a snapshot is taken of the current execution state and the program is allowed to continue execution. When the execution terminates or hits an exploration limit, one of the previously captured snapshots is restored and a different decision outcome is forced by overriding the program counter [125, 8]. As a result, forced execution is imprecise: resulting execution states may be infeasible, causing analysis to explore execution paths that never occur during real execution. Understanding the trigger conditions is usually out of scope for these approaches. More advanced examples of forced execution try to improve precision by employing taint analysis [82, 97] to force only the predicates that are related to program input.

A more precise analysis can be achieved by using *dynamically configured environments*. These approaches execute the program in different controlled environments, typically sandboxes, compare the execution results, and search for mismatches. Setting up the controlled environments is usually semi-automated. Well-known API return values or events in the system can be supplied to the program automatically [41, 58, 116, 126, 22, 23], however, persistent objects need to be set up by human analysts [1, 74]. This analysis method is more precise than forced execution because the analyzed execution paths are feasible. However, correctly setting up the environment such that all the right trigger conditions are met during execution remains a major challenge.

The most precise, yet least scalable approaches use a variant of *symbolic execution*. These approaches model the environment with symbolic summary functions, which introduce symbolic variables to analysis. Concrete trigger inputs are acquired by taking

the path conditions calculated by the symbolic execution engine and solving them using a constraint solver. The main limitation of current approaches, however, is that they consider only a limited number of trigger condition types in order to contain path explosion. MineSweeper [16, 132] relies on the human analyst to select trigger condition types. TriggerScope [43] only considers time, location and SMS objects. Rozzle [65] limits itself to the navigator's fields in JavaScript. BotMelt [59] focuses only on network packet data and Li *et al.* [72] allow certain environment-related data to be returned concretely for performance reasons.

The main difference between our work and the previously discussed related work is that we expand on previous work relying on symbolic execution by considering all potential trigger condition types automatically. As a result, our analysis introduces more symbolic variables, therefore making the path explosion problem a bigger threat to the success of our analysis. In order to overcome this challenge, our approach focuses on searching for specific instances of malicious behavior, which we model differently at the source code and binary levels.

4.2.2 Reproducing field failures

The software engineering problem of reproducing field failures has a similar goal as our present work. In that field, the assumption is that the user experiences a fault or crash and files a bug report. Then, it is the developers' task to debug and fix the issue, which requires triggering the same fault or crash in a controlled environment. In order to help the developers, numerous approaches have been proposed to automatically force (trigger) the program to malfunction. These approaches assume that some information is available about the crash, e.g., specific program points, the call stack at the time of the crash, function calls and/or UI events leading to the crash, complete traces, or parts of a log file.

Approaches based on symbolic execution, such as BugRedux [56], F3 [57], Hercules [100], RDE [123], and STAR [26], employ a symbolic execution engine to automatically generate test cases for developers to observe the malfunction in their debugging environment. In order to deal with the potentially large number of execution paths, these approaches use the information about the fault/crash to guide symbolic execution.

Approaches relying on observed events leading up to the malfunction, such as Pensieve [134], Yakusu [40], RETracer [35], and CRASHDROID [124], utilize program analysis techniques, e.g., dependency analysis or taint analysis. These approaches typically scale better than symbolic execution-based approaches but can often only produce partial traces ending at the fault/crash site.

Search-based approaches, e.g., EvoCrash [113, 114], SBFR [63], and RECORE [105], use genetic algorithms to generate test cases for developers. These approaches formulate reproducing field failures as an optimization problem and define custom fitness functions. Candidate test cases are generated using crossover and mutation operations. Defining the fitness function is one of the major challenges because it involves measuring the distance between a test case and the targeted program point. There exist other approaches as well, including crowd-sourcing the problem of acquiring information about the mal-

function [48], exploring the program's state space with model checking [86, 87] and using random testing to generate the test cases [127, 14].

The major difference between the aforementioned works and our work is the amount of input data available. Approaches to reproduce field failures assume the availability of execution data about an existing fault/crash. In other words, they can assume that the failure definitely exists in the application and there certainly is a way to make the program exhibit the malfunction. What is more, the available execution data can be used to pinpoint the location of the crash and guide the search for a test case. By contrast, we do not know whether a malicious behavior exists in a program or not. We also do not have *a priori* information about where such a behavior could be located. We can locate invocation instances of library and system calls in the interprocedural control-flow graph, however, we do not know whether a particular invocation instance contributes to the malicious behavior or not. We also do not have (partial) information about how to force the program to exhibit the malicious behavior.

4.3 Preliminary study at the source code level

Before moving to the binary level, we perform a study on the applicability of our approach at the source code level. There are two rationales behind performing this experiment. First, trigger-based behavior may be present at the source code level during development. For example, a disgruntled employee[3] or a compromised supplier may deliver code with hidden unwanted functionalities. These scenarios show that uncovering trigger-based behavior is also a software verification and validation problem: given a piece of software that conforms to its specification, is said piece of software capable of performing other functionalities as well? And if so, what are the ramifications of those other functionalities? Second, program analysis at the binary level is more challenging than it is at the source code level. If there are fundamental issues with the idea of replacing all external data with symbolic variables, those issues are more easily understood from the source code than from binary instructions.

For our experiment, we follow the methodology outlined in Fig. 4.2. For each piece of software under analysis, we automatically instrument its source code to introduce fresh symbolic variables instead of concrete environmental data. We then subject the instrumented program to mixed concrete and symbolic execution, which generates test cases for individual execution paths. By refeeding the generated test cases to the software under analysis, we acquire the execution trace of an execution path. We examine this execution path for signs of malicious or undocumented behavior.

4.3.1 Automatic source code instrumentation

Many existing mixed concrete and symbolic execution tools, including KLEE [18], S2E [27], and CREST [17], require additional library calls to specify which variables in the software

[3] http://www.computerworld.com/article/2551740/government-it/it-worker-jailed-for-creating-logic-bomb.amp.html (Last visited: Jan 4, 2021)

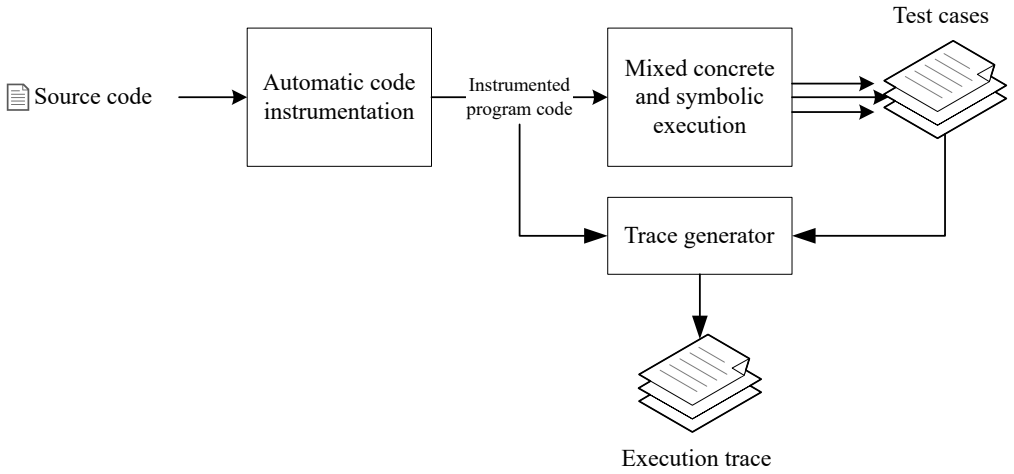

Figure 4.2: Methodology for evaluating open-source software with respect to trigger-based behavior

should be treated as symbolic. Therefore, the first step of the analysis is to instrument the source code with the required library calls.

In order to detect trigger-based behavior, instrumentation must first identify variables and function calls that may guard the hidden behavior. Considering that the attacker triggers the hidden behavior 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 *symbolic summary functions*. Symbolic summary functions are empty functions that 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. Symbolic summary functions should be easily identifiable in the source code, e.g., with naming convention, to make the the introduction of symbolic values 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: it could be called `pcap_next_dummy`. However, instead of returning a pointer to a concrete packet, the symbolic summary function returns a pointer to a symbolic variable representing a packet.

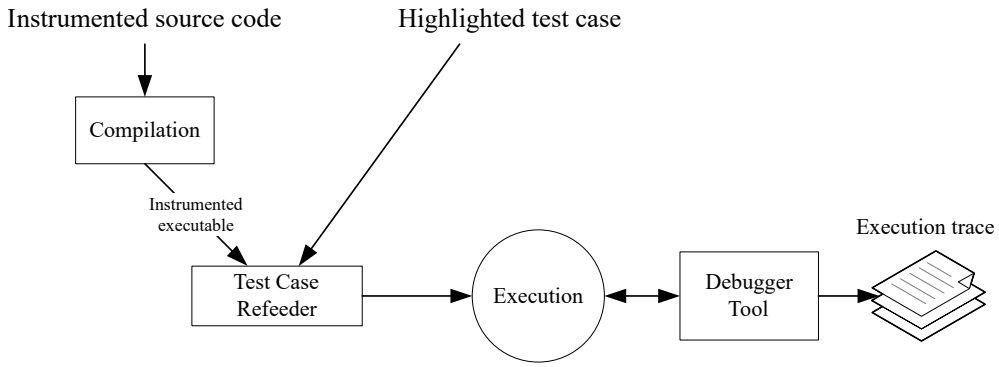

Figure 4.3: Generating execution traces from highlighted test cases

4.3.2 Mixed concrete and symbolic execution

The mixed concrete and symbolic execution tool then analyzes the instrumented source code and 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, and
2. path conditions output at potentially malicious points must be easily differentiated from other test cases generated at different program points.

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 between generated test cases. 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.

4.3.3 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

Table 4.1: Tools used in the prototype implementation of the methodology

Framework Element	**Tool in Prototype**
Automatic source code instrumentation	clang [69]
Mixed concrete and symbolic execution	KLEE
Compilation	gcc[4]
Test case refeeder	klee-replay
Debugger	gdb[5]

generation. We 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. 4.3.

First, the instrumented source code has to be compiled into an executable, which contains the symbolic summary 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. Human analysts can then inspect the execution trace to determine whether the execution path is indeed malicious or not.

4.3.4 Prototype implementation

We implemented our methodology as a prototype using the GNU and LLVM toolchains using the tools summarized in Table 4.1. Automatic source code instrumentation was implemented as a standalone tool using clang. Our prototype automatically generates symbolic summary 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 symbolic summary functions:

```
{
    "function_name"         : "recv",
    "include_file"          : "sys/socket.h",
    "symbolic_return"       : false,
    "symbolic_return_size"  : 0,
    "symbolic_params"       : [
        {
            "index"             : 1,
```

[4] https://gcc.gnu.org/ (Last visited: Jan 4, 2021)
[5] https://www.gnu.org/software/gdb/ (Last visited: Jan 4, 2021)

```
            "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 are 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 4.3.2. 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, thereby forcing KLEE to calculate and output the potential trigger inputs. 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.

4.3.5 Evaluation

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 and used its default path selection strategy.

Table 4.2: 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

Our results are summarized in Table 4.2. 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 command line 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 command line 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, i.e., the sample executes concurrently. However, KLEE cannot analyze concurrent execution, leading to a failed analysis.

4.4 Tackling the problem at the binary level

The experiments at the source code level show that the default path selection strategy does not always yield positive results. Therefore, we need a path selection strategy that guides symbolic execution towards specific program points of interest. We now propose a new path selection strategy that fits this criteria and can be used to analyze binaries. At the binary level, we assume that the analyzed program is deterministic and interacts with the environment through the operating system and its API (system calls). Therefore, we consider invoked library functions as part of the analyzed program.

We assume that the human analyst is interested in knowing the conditions required to trigger a specific malicious behavior in the analyzed program. However, as we mentioned before, we assume no *a priori* information about whether the behavior in question exists in the binary, and if it does, where it could be located in the binary. To guide our search, we require 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 chapter, we 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 for a number of reasons. First, system calls constitute the primary communication channel between programs and the operating system. 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 [19]. Therefore, we 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. For example, if analysis focused on checking whether the analyzed program spoofs the source IP address of outgoing packets during a denial-of-service attack, the target system call pattern could contain the following:

(`socket`, `type`=`SOCK_RAW`),
(`sendto`, `sockfd`=`raw_sock_fd` $\wedge$ `buf[SRC]` $= x$),
(`sendto`, `sockfd`=`raw_sock_fd` $\wedge$ `buf[SRC]` $\neq x$),

1. `socket`: The `socket` system call[6] is used to create sockets for network communication, including denial-of-service attacks. In order to spoof the source IP address in the IP header, the program would need to create a raw socket. Raw sockets expect the IP header and transport layer header to be constructed by their callers[7]. As a result, the associated predicate would check whether the `type` argument of the system call requests a raw socket.

2. `sendto` at least twice: The `sendto` system call[8] can be used to send data over sockets, including the payload during denial-of-service attacks. The associated predicate would need to perform two checks: 1) determine whether the data is sent over the previously created raw socket, and 2) check whether the bytes representing the source IP address have different values across system call invocations.

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[9]. 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.* [79].

The goal of our analysis is to find an execution path that reaches the specified system calls in the same order as specified in the target system call pattern and whose execution states at the system call invocations satisfy the associated predicates. The execution path may contain other system calls between the elements of the target system call pattern. In order to find such an execution path, we use mixed concrete and symbolic execution guided by Algorithm 3. Our approach consists of three techniques:

1. a path selection strategy, executed at line 7, consisting of three levels to prioritize available execution paths,

2. symbolic summary functions, used at line 9, capturing the behavior of invoked system calls in order to introduce a model of environmental data to the analysis, and

[6] http://man7.org/linux/man-pages/man2/socket.2.html (Last visited: Jun 8, 2020)
[7] http://man7.org/linux/man-pages/man7/raw.7.html (Last visited: Jun 8, 2020)
[8] http://man7.org/linux/man-pages/man2/sendto.2.html (Last visited: Jun 8, 2020)
[9] https://virustotal.github.io/yara/ (Last visited: Mar 5, 2021

Algorithm 3 Determining Trigger Conditions

Input:
icfg: interprocedural CFG of program
summaries: symbolic summary functions
pattern: ordered list of system calls and predicates $(s_0, p_0), (s_1, p_1), \ldots, (s_n, p_n)$
Output:
path_conds: triggering execution paths' path conditions

```
1:  begin
2:      path_conds ← []
3:      st_0 ← execution state at program's entry
4:      states ← []
5:      append(states, st_0)
6:      while states is not empty do
7:          to_step ← PathSelection(states, icfg, pattern)
8:          remove(states, to_step)
9:          successors ← Step(to_step, summaries)
10:         for all st in successors do
11:             UpdateICFG(st, icfg)
12:             ▷ PatternCheck() saves triggering path condition in path_conds
13:             path_conds ← PatternCheck(st, icfg, pattern, path_conds)
14:             insert(st, states)
15:         end for
16:         ▷ Some of successors may have triggered pattern
17:         if path_conds contains path condition(s) then
18:             return path_conds
19:         end if
20:     end while
21:     return null
22: end
```

3. a mechanism for automatically finding target program points that can advance the execution path's progress in the target system call pattern, invoked at line 13.

We start analysis 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 discussed in Section 4.4.1, see lines 2 to 7. The selected state(s) are symbolically analyzed by invoking mixed concrete and symbolic execution. In order to model the environment, we also supply custom symbolic summary functions, discussed in Section 4.4.2, at line 9. After each *Step*, we check whether the target system call pattern has been triggered in the procedure *PatternCheck* (line 13), and select control-flow graph nodes to be reached depending on the results. The inner working of this procedure is discussed in Section 4.4.3.

4.4.1 Path selection strategy

Symbolic summary functions introduce only the model of environmental data in the form of fresh symbolic variables. The actual conditions required to trigger a specific behavior in the analyzed program are encoded in its instructions. In order to calculate the correct environmental values, we need to recover and solve these conditions. To this end, we use mixed concrete and symbolic execution, capable of both recovering these conditions as path conditions and solving them using Satisfiability Modulo Theory solvers.

In order to overcome the path explosion problem, we employ a path selection strategy with three levels. At the first level, we prioritize execution states based on how far they have progressed in the target system call pattern, i.e., how many system calls with correct predicate they have already executed. This is a greedy strategy as we analyze execution paths closer to completing the target system call pattern first. If there is exactly one execution state which has progressed further than all others, we select that state for further analysis.

If prioritization based on progress in the target system call pattern yielded multiple execution states, we move on to the second level. At this level, we further prioritize between the states with the same priority based on the expected calling context in which they are likely to reach the next element of the target system call pattern. Note that different execution states progress differently with respect to the target system call pattern and as such, available execution states may aim to reach different invocation instances of system calls. Because we have no information about the calling contexts of system calls necessary to trigger the target system call pattern, we aim to explore as many calling contexts as possible. Execution states that are expected to reach invocation instances in previously unseen calling context are analyzed first. The rationale behind this strategy is that invocation instances in unseen calling contexts can reveal a wider range of behaviors implemented in the program, especially, if no invocation instance uncovered so far could satisfy the associated predicate. If there is exactly one execution state which is expected to produce a new calling context, we select that state for further analysis.

If path selection based on expected calling context yielded multiple execution states to follow, we move on to the third and last level. At this level, we employ shortest-distance symbolic execution (SDSE) [77], designed to prioritize execution paths which are closer to a selected target according to some metric. There may be scenarios, in which multiple states share are equally close to a selected target. In such cases, we analyze all such execution states in parallel. SDSE was originally proposed to solve the line reachability problem: how to reach a target line in the source code? It requires the interprocedural control-flow graph in order to guide symbolic execution towards the targeted line. The strategy first translates execution paths to control-flow graph nodes, and then computes the shortest distance from said nodes to the node corresponding to the target line. The computed metric is used as scores to prioritize execution paths. At branches, SDSE selects the execution path with the lowest score among all available paths for analysis.

Our scenario is similar to the one SDSE was developed for in the sense that we need

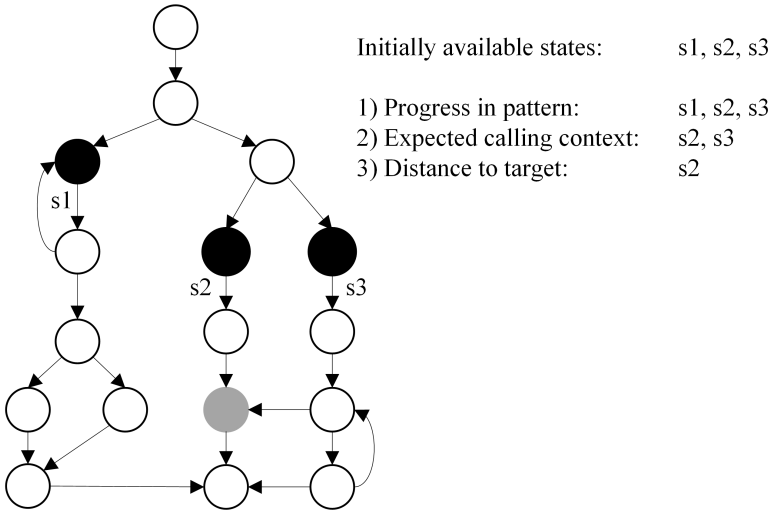

Figure 4.4: Example scenario for our proposed path selection strategy

a solution for the reachability problem in order to reach invocation instances of system calls and check the associated predicates. However, there are key differences as well. First, SDSE was originally proposed and implemented at the source code level, while we apply it at the binary level. As a result, instead of a target line, we aim to reach a target binary instruction. Secondly, as stated in [77], SDSE can only work correctly, if the recovered interprocedural control-flow graph does not have mismatching calls and returns. Otherwise, semantically incorrect or infeasible paths may be computed as shortest paths, resulting in incorrect scores and priorities. In order to generate a semantically correct control-flow graph whose structure properly captures function calls and returns encountered during execution, the generator algorithm has to consider a lot of context-related information, including call sites, return sites and the call stack. There exist algorithms capable of handling that information [21, 111], however, their usage in practice poses a challenge. The more context-related information is taken into consideration, the more time and space are required to generate and store the resulting control-flow graph, the increase is exponential. Therefore, we implemented a heuristic algorithm that can discard those edges of an approximate interprocedural control-flow graph whose inclusion in the shortest path calculation might result in incorrect paths. We do this by simulating changes to the call stack during shortest path calculation wherever edges representing function calls and returns are encountered. Our heuristic is also capable of adding missing edges and nodes to the control-flow graph based on the execution states mixed concrete and symbolic execution produces (procedure $UpdateICFG$ at line 11 of Algorithm 3). As a result, we can keep the required contextual information and we can employ generic shortest-path calculation algorithms to generate semantically correct shortest paths. We discuss implementation of the heuristic in Section 4.5.

Figure 4.4 shows an example interprocedural control-flow graph in order to demonstrate our path selection strategy. Let us assume that mixed concrete and symbolic exe-

cution has produced states `s1`, `s2` and `s3`, which are translated to the black control-flow graph nodes. Let us also assume that the grey node represents an invocation instance of the first system call in the target system call pattern and that no previous calling context has been observed for this system call. Our path selection strategy would narrow down the states to analyze as follows. First, as no state has satisfied the grey node's associated predicate, the first level determines that all available states are progressing towards the first element in the target system call pattern. As a result, prioritization based on progress in the target system call pattern does not eliminate states. Then, prioritization based on expected calling context removes `s1` from the set of available states, because it cannot produce an expected calling context (it cannot reach the grey node). At last, `s2` is selected because its translated control-flow graph node is closer to the grey node (2 steps), than the node representing `s3` (3 steps).

4.4.2 Symbolic summary functions

As mentioned previously, the environment is represented by operating system services, and the environment manifests itself as the result of invoking system calls. Therefore, we need symbolic summaries of system calls, which model their effects. Such summary functions allow us to simulate the environment for the analyzed program and enable mixed concrete and symbolic execution to analyze how returned data influences execution.

Our summaries are semantically equivalent to the system calls they replace with two major exceptions. First, if the system call writes into the environment (e.g., sends packets or writes in a file), the summary always returns with success. This allows us to contain the path explosion problem: if we simulated both success and failure, we would need to simulate the various conditions for failure, which would further increase the number of execution paths to analyze. We acknowledge the possibility of system call failures being used as triggers. Should the analysis infrastructure be strong enough to handle the increased number of execution states, system call failures could also be modeled using symbolic summary functions. Secondly, if the system call returns data from the environment (e.g., assigned process ID, system time, network messages), the summary function returns fresh symbolic variables instead. Using the fresh symbolic variables, the influence of the environment on the program can be analyzed.

Symbolic summaries can be written based on the semantic information available about the system calls in the operating system's documentation. While expertise is required to write them, they need to be written only once for a particular platform. As an example, let us consider the Linux system call `fork`, responsible for duplicating processes. On success, `fork` returns the PID of the child process in the parent and 0 in the child. On failure, it returns -1 to the parent, creates no child process and sets the global variable `errno` appropriately. In order to explore how the invocation of `fork` influences the analyzed program, we need to replace its return value with a fresh symbolic variable. According to its manpage[10], its return value has the type `pid_t`, a signed integer. On the ARM platform, a signed integer is 32 bits long; therefore, the

[10] http://man7.org/linux/man-pages/man2/fork.2.html (Last visited: Jun 8, 2020)

model of this system call must return a 32-bit long symbolic variable for that particular platform. The variable must be constrained as written in the documentation: it can be a positive number, 0 or -1. Two further constraints must be added to the model to capture its behavior faithfully. First, if the return value is greater than 0, then semantically, analysis continues in the child process, and the PID, as well as the parent PID of the execution state must be updated accordingly. Secondly, if the return value is -1, then semantically, the system call failed and a new symbolic variable is required to represent the error condition whose value must be constrained to one of the potential error codes.

4.4.3 Selecting control-flow graph nodes of interest

As mentioned before, different execution states at any given point in the analysis progress differently with respect to the target system call pattern. Each system call in the target system call pattern can have multiple invocation instances in the analyzed program; therefore, many combinations of invocation instances are available for analysis. However, only certain combinations can satisfy the associated predicates.

When an execution state reaches the invocation instance of a system call, the associated predicate must be checked. There are two possible outcomes to consider depending on the result as described by Algorithm 4. If the associated predicate cannot be satisfied, a different invocation instance or a different path to the same invocation instance must be selected. Therefore, we backtrack to an execution state (line 6 of Algorithm 4) at which the previous system call's associated predicate was satisfied and select another invocation instance of the currently targeted system call (line 7 of Algorithm 4). If the associated predicate of the first element in the target system call pattern cannot be satisfied, the execution state must be backtracked to the analyzed program's entry point.

If the associated predicate can be satisfied, the execution state is one step closer to completing the target system call pattern, potentially even completing it. If the target system call pattern is completed, we can stop analysis and output the path condition and a satisfying assignment to the analyst. Therefore, we save the path condition to *path_conds* in line 12 of Algorithm 4. If the target system call pattern is not completed, we must select an invocation instance of the next system call in the target system call pattern. We also save a copy of the execution state in order to support the previously discussed backtracking with respect to the target system call pattern, see lines 15 and 16 of Algorithm 4.

When selecting an invocation instance of any system call, we need to take into consideration the context sensitivity of the interprocedural control-flow graph. If its context sensitivity is high enough, the graph can include multiple nodes that represent the invocation of the same system call in different calling contexts. However, if its context sensitivity is low, different invocations of the same system call can be captured in the same node. Given the system call target and the set of control-flow graph nodes representing said system call, we choose between the control-flow graph nodes using the following strategy:

Algorithm 4 *PatternCheck*

Input:
st: execution state (successor after ivoking *Step*)
icfg: interprocedural CFG of program
pattern: ordered list of system calls and predicates $(s_0, p_0), (s_1, p_1), \ldots, (s_n, p_n)$
path_conds: list of triggering execution paths' path conditions
Output:
path_conds: modified with *st*'s path condition, if *st* completes *pattern*

```
1: begin
2:     if st at s_i then
3:         call_sites ← null
4:         pred ← CheckPredicate(st, p_i)
5:         if pred is False then
6:             st ← GetCopy(s_{i-1})
7:             call_sites ← GetInvocationSites(s_i)
8:         else
9:             if s_i = s_n then
10:                ▷ PatternCheck is invoked for all successors generated by Step
11:                ▷ → Multiple successors may trigger pattern
12:                append(path_conds, st.path_cond)
13:                return
14:            else
15:                SaveCopy(st, s_i)
16:                call_sites ← GetInvocationSites(s_{i+1})
17:            end if
18:        end if
19:        SetupTargetNodes(st, call_sites)
20:    end if
21: end
```

- If there exists a control-flow graph node representing the invocation of said system call which has not been visited before, then that node represents a previously unseen calling context for said system call. We select that node to be reached using SDSE.
- If only visited nodes are available, we construct semantically correct shortest paths in the control-flow graph which end with the visited nodes, and check whether following those paths yield an as yet unseen calling context. We achieve this by simulating changes to the call stack along the generated paths.

We check semantically correct shortest paths in the following order. First, we check the expected calling contexts along the shortest paths between the control-flow graph node representing the current execution state and the visited nodes. If all result in expected

calling contexts previously observed, we iterate over the call sites of the visited nodes, forcibly including them in the semantically correct shortest paths and checking their expected calling contexts. If no unseen expected calling context is found, we include the call sites of the forcibly included call sites in the semantically correct shortest paths and recheck them as before. We iterate over the call sites of previously analyzed call sites until an unseen expected calling context is discovered. Then, we select the visited node, as well as all control-flow graph nodes representing the beginning of the functions in the expected calling context as nodes to be reached and calculate the priority of the state with respect to this path during path selection.

4.5 Prototype implementation for the binary level

We implemented our approach in angr (version 7.8.2.21), an open-source binary analysis tool written in Python, capable of analyzing binary formats of major operating systems, such as ELF, PE and Mach-0 files. The tool implements many analyses for binary code, including mixed concrete and symbolic execution, constraint solving, control-flow graph generation, program slicing, dependency analysis, etc. These analyses are performed over the intermediate representation (IR) of valgrind [88], called VEX, to provide platform independence. VEX translates a sequence of binary instructions into a block of IR instructions. As a result, most analyses are not performed on a per instruction basis, but rather on a per IR block basis. Our implementation uses the following features of angr:

1. mixed concrete and symbolic execution engine with a constraint solver,
2. control-flow graph generation, and
3. model of execution states, including registers, memory, and elements from POSIX, such as files and sockets.

To implement our methods described in the previous section, we modified the workflow and execution of angr at certain places. We discuss these modifications in this section.

4.5.1 Symbolic summaries for system calls

angr supports system call invocations during mixed concrete and symbolic execution. However, developers focus more on defining the environment at the library level and therefore, the tool has more symbolic summaries for standard libc functions than it has for system calls. As a result, many system calls invoked during our tests were missing and had to be added to the tool manually. The list of 41 system calls we had to create symbolic summaries for is shown in Table 4.3. In order to contain the path explosion problem, our symbolic summaries for `read` and `recv` have a configurable upper limit set for the length of returned data.

Table 4.3: System calls on Linux for which symbolic summaries were created

_newselect	arm_set_tls	brk
clone	close	connect
dup2	execve	exit
exit_group	fcntl	fcntl64
fork	futex	geteuid32
getgid32	getpid	getppid
gettimeofday	getuid32	ioctl
kill	mmap2	nanosleep
open	pipe	read
recv	rt_sigaction	rt_sigprocmask
sendto	setrlimit	setsockopt
socket	sysinfo	time
ugetrlimit	uname	vfork
wait4	write	

4.5.2 Extending the interprocedural control-flow graph

There are two algorithms to recover the interprocedural control-flow graph in angr. The first algorithm is CFGFast and it relies on heuristics and assumptions to greatly decrease the time required for generation. The second algorithm is called CFGAccurate (CFGEmulated in later versions) and it performs lightweight symbolic execution for control-flow recovery, increasing accuracy. In our implementation, we used CFGAccurate as accuracy is important for using SDSE.

There are program constructs that pose a challenge during control-flow graph generation, e.g., indirect jumps. We encountered scenarios where CFGAccurate detected the indirect jumps but it was unable to determine the address the analyzed code jumped to accurately. The limitation is caused by the lightweight nature of its symbolic execution: if a read or write operation involves an operand which could be assigned multiple values, that operand is skipped and a fresh, unconstrained symbolic variable is used instead. However, angr's symbolic execution has an upper limit on the number of successor states it generates when analyzing an execution state. If the instruction pointer of the analyzed execution state has more than 256 solutions (by default), then the tool assumes that the instruction pointer was overwritten with unconstrained data, and flags the execution state as one producing unconstrained successors.[11] As a result, CFGAccurate may fail to analyze certain parts of the program due to the inaccurate execution state used during construction. This scenario is illustrated with the following two instructions:

```
; load function address from memory
ldr r4, [r3, #4]
```

[11]This assumption is included in angr's documentation together with the fact that it is not sound in general.

```
; call function
blx r4
```

The code includes a call to the address contained in `r4`, whose value is loaded from memory. The address from where the value is to be loaded is influenced by `r3`. If `r3` holds an operand with multiple potential values while control-flow recovery analyzes this code segment, then analysis has to read a multi-valued operand from the register. However, as discussed before, instead of performing the read, the recovery algorithm creates a fresh, unconstrained symbolic variable to represent the result of the read operations. As a result, `r4` will also hold an unconstrained symbolic variable when the recovery algorithm tries to determine the jump address. Because the unconstrained symbolic variable has more than 256 solutions, the state is flagged as one producing unconstrained successors and address resolution fails.

Normal mixed concrete and symbolic execution, however, never skips operands and is less likely to run into such a scenario. Execution states have operands with semantically correct values and correct path constraints. When control-flow recovery is resumed from such a state, CFGAccurate can accurately identify the indirect jump addresses, if the value of `r4` has less than 256 solutions. Therefore, during control-flow recovery, we take note of addresses where unconstrained successors were computed as potential extension points of the control-flow graph. When mixed concrete and symbolic execution reaches such an address, we use the accurate execution state to extend the control-flow graph on the fly.

angr includes other performance-increasing heuristics which affect the accuracy of the generated control-flow graph, including limits on the number of times a block of instructions must be analyzed during control-flow graph generation. This setting affects the recovery of loops as well as function return addresses. The latter is also affected by the configured context sensitivity: if a function is encountered in a calling context in which it has been analyzed before, it will not be analyzed again. As a result, return edges may be missing from the control-flow graph. In order to address this challenge, each time mixed concrete and symbolic execution descends into a function, we statically check whether the control-flow graph contains the correct return edge. If the edge is missing, we extend the control-flow graph by adding it.

4.5.3 Shortest path calculation in the control-flow graph

The level of context-sensitivity influences the accuracy of CFGAccurate. This parameter captures how deep the call stack is taken into consideration when determining the calling context of any given control-flow graph node. By default, the algorithm analyzes each address only once per distinct calling context. As a result, different levels of context sensitivity result in different graph structures, which in turn influence the available paths computed by generic shortest path algorithms. Figure 4.5 shows the different contexts in which functions are analyzed at different levels of context sensitivity. Because of the different contexts, functions may be replicated multiple times in the control-flow graph. Note, that we demonstrate the effect of context sensitivity at the source code level only

```
void c(){
   printf("Executing function c");
}
void b(){
   printf("Executing function b");
   c();
}
void a(){
   printf("Executing function a");
   c();
}
int main(int argc, void* argv) {
   a();
   b();
   return 0;
}
```

	0-context sensitivity	1-context sensitivity	2-context sensitivity
a	a	main→a	(lib init)→main→a
b	b	main→b	(lib init)→main→b
c	c	a→c b→c	main→a→c main→b→c
printf	printf	a→printf b→printf c→printf	main→a→printf main→b→printf a→c→printf b→c→printf

Figure 4.5: Different contexts of functions during control-flow graph generation

for ease of understanding, but our techniques work at the binary level.

In order for generic shortest path algorithms to compute semantically correct paths in the interprocedural control-flow graph, edges connecting mismatched call sites and return sites must be discarded. CFGAccurate annotates edges with attributes recovered by VEX during lightweight symbolic execution, including the semantics of the jump at the end of each IR block (e.g., function call, return, etc.). It also saves some of the frames of the call stack in which the control-flow graph node was originally recovered. The depth of the saved stack is specified by the context sensitivity level: at context sensitivity level 1, the upper most frame is saved, at context sensitivity level 2, the two upper most frames are saved, etc. Inspired by the control-flow graph model of Babić *et al.* [7], a visibly push-down automaton, which keeps track of the calling context of functions, we simulate the call stack along a given execution path using the following heuristics:

1. Initially, the simulated call stack is equivalent to the execution state's call stack.

2. If along the path, an edge's attributes suggest a function call, we simulate it by pushing a new frame upon the simulated call stack, saving the call site's address, the called address and the address where the function is supposed to return (return target).

3. If along the path, an edge's attributes suggest a function return, we check whether the address of the edge's destination control-flow graph node matches the recorded return target and whether the saved call stack frames at the edge's destination control-flow graph node match those of the simulated call stack. If either of the checks fails, we discard the edge as semantically incorrect.

4. If the edge's source and destination control-flow graph nodes have the same saved

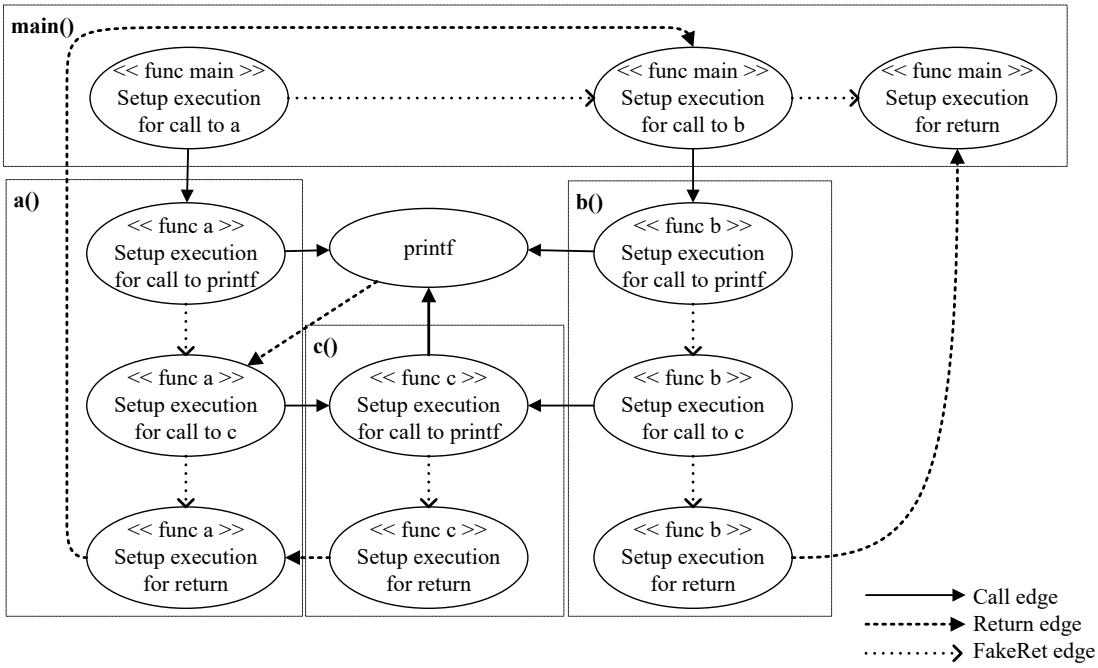

Figure 4.6: Fake return edges in an interprocedural control-flow graph

call stack frames and the edge's attributes suggest neither a function call, nor a function return, no changes are made to the simulated call stack.

Thanks to the above rules, we are able to generate semantically correct, context-aware shortest paths using generic shortest path algorithms.

Another important feature of `angr`'s control-flow graph recovery is the use of the so-called *fake return* edges. These edges are directed edges connecting a function's call site to its return site and are automatically added by angr whenever a call is encountered. Fake return edges essentially represent the execution of a function, abstracting the actual nodes and edges of the function away. Figure 4.6 depicts the fake return edges in the interprocedural control-flow graph of the source code shown in Figure 4.5 when context sensitivity level is set to 0. For the sake of clarity, the actual instructions responsible for setting up the execution state for calling functions were omitted. By default, CFGAccurate analyzes each IR block once per distinct calling context. At context sensitivity level 0, the calling context is limited solely to the currently analyzed function, which leads to each function being present in the graph exactly once. For each analyzed block, `angr` adds a call edge to the called function and a fake return edge to the return site. These special edges mainly serve the purpose of ensuring connectivity in the graph. Because each block is analyzed once per distinct calling context, each function has only 1 return edge. For example, consider the `printf` function in Figure 4.6. Even though it is called from `a`, `b` and `c`, it is analyzed only once, the first time it

is encountered when called from `a`. As a result, `printf` has only 1 return edge, leading to its return site in `a`. Without fake return edges, `printf`'s call site in `b` would not be connected to its return site in `b`.

Without fake return edges, our edge discarding heuristic would be unable to calculate certain paths in the control-flow graph. For example, our heuristic discards the return edge between `printf` and its return target in `a`, if the execution state is executing in `b` at the time of shortest path calculation. Because the semantically correct return edge connecting `printf` to its return site in `b` is missing from the graph and the available return edge is discarded, generic shortest path algorithms must rely on the fake return edge to calculate shortest paths. In reality, however, the execution of the actual function (`printf` in this case) must be simulated. In order to faithfully capture the cost of calling a function, we assign weights to fake return edges equal to smallest number of IR blocks simulated between the call and return sites throughout analysis, i.e., the shortest path mixed concrete and symbolic execution uncovered. We also assign the same weight to call edges as well, causing shortest path calculation to favor fake return edges. If call edges were not weighted, shortest path calculation would prefer descending into the function and using the shortest path inside the function, which may not correspond to its real simulation cost because of, for example, loops. Our weighting technique, allows us to consider the actual simulation cost of functions based on previous traversals. Thanks to this heuristic, we can keep context sensitivity at level 1.

4.5.4 Call stack management

During our work, we discovered mismatches between how the call stack is managed in CFGAccurate and how it is managed during mixed concrete and symbolic execution. The discrepancies between the algorithms initially hindered us in translating execution states into control-flow graph nodes.

In case of mixed concrete and symbolic execution, function calls are detected by statically looking at the semantic information about the jump at the end of the analyzed IR block. Function returns, on the other hand, are detected by looking at the stack pointer. The function returns if either the stack pointer has a lower value than it had at the call (which is the convention e.g., on Intel platforms), or execution has reached the return address recorded at the call and the stack pointer has the same value as it had at the call (which is the convention in platforms like ARM where the return address is stored in the link register).

CFGAccurate uses the same method with an additional feature. For each IR block address encountered during CFG construction, it checks with angr's loader whether the address corresponds to a symbol. If it does, it forcefully simulates a call to that symbol. This method has the advantage of providing more meaningful nodes in the control-flow graph. However, it hinders us from accurately matching execution states to control-flow graph nodes as the calling contexts are different. As an example, consider the following instructions:

```
000105a4 <getspoof>:
```

```
   ...
   105bc: eb0022aa  bl 1906c <rand>
   ...
0001906c <rand>:
   1906c: ea000065  b 19208 <__GI_random>
   ...
00019208 <__GI_random>:
   ...
```

The `getspoof` function at `0x105bc` calls `rand`, which immediately jumps to `__GI_random`. In case of symbolic execution, the execution state at `0x19208` has the calling context `getspoof`→`rand`, while the control-flow graph node representing `0x19208` has the context `getspoof`→`rand`→`__GI_random`, because `0x19208` corresponds to a symbol. Due to the different calling contexts, the execution state cannot be translated to the control-flow graph node. Thus, we removed the forceful simulation of function calls from CFGAccurate.

We have also encountered call stack management issues in scenarios where mixed concrete and symbolic execution forks in functions with only one of the paths returning. The issues are caused by angr running its call stack management code before adding path constraints to the state. We illustrate the problem with an example. Consider the following snippet from the `strcasecmp_l` function.

```
   ; ip points to next character in string1
   179b8: e5dc3000  ldrb r3, [ip]
   ; r1 points to next character in string2
   179bc: e5d10000  ldrb r0, [r1]
   ; case-insensitive comparison -> transform characters
   ; transformation depends on current locale (r2)
   179c0: e592e004  ldr lr, [r2, #4]
   179c4: e1a03083  lsl r3, r3, #1
   179c8: e1a00080  lsl r0, r0, #1
   179cc: e19e30f3  ldrsh r3, [lr, r3]
   179d0: e19e00f0  ldrsh r0, [lr, r0]
   ; lower-case character comparison and conditional return
   179d4: e0530000  subs r0, r3, r0
   179d8: 149df004  popne {pc} ; (ldrne pc, [sp], #4)
   179dc: e4dc3001  ldrb r3, [ip], #1
```

The function iterates over two strings character by to check whether they are equal (case insensitive). The comparison between two characters is implemented using subtraction. If the result of the subtraction is 0, i.e., the characters are the same, then the function continues, otherwise, it returns. If any of the input strings consists of symbolic variables as characters, the comparison has two outcomes: equals and not equals. At the end of simulating the block at `0x179b8`, angr forks and creates the two successor states, one at `0x179dc` and another at the return site. It then proceeds to check whether any of these states returned from `strcasecmp_l`. However, the path condition has not been added to the successors at the point yet, therefore, the stack pointer of the state

at the return site is a symbolic expression encoding both staying in the function and returning. As a result, the call stack management code cannot deduce the return and fails to pop `strcasecmp_l` from the call stack. To overcome this issue, we concretize the stack pointer after forks and re-run the call stack management code to get correct call stacks.

4.5.5 Model of the execution state

In order to model the side effects of system calls and any additional data they might return, we extended the original execution state model provided by angr. The extended model includes additional POSIX elements on a per-path basis, such as group ID, thread ID and parent process ID.

We also modified how system time is tracked throughout mixed concrete and symbolic execution. Originally, angr used a monotonically increasing, global symbolic variable to model system time which is suitable for the default breadth-first exploration strategy. However, SDSE's prioritization strategy can backtrack to an earlier execution state, which semantically means taking us "back in time". In order to support such a backward flow of time, we model system time on a per-path basis with local symbolic variables.

Throughout mixed concrete and symbolic execution, we also monitor the execution state to detect whether branches are the result of references to uninitialized memory addresses. This scenario could be the result of a bug in the analyzed program, but might also signal missing side effects of system call models. Therefore, we do not pursue such paths any further, but keep them separated from the rest of execution states for further analysis.

4.5.6 Keeping track of execution states' progress

In order to track each execution state's progress with respect to the target system call pattern and the selected control-flow graph nodes they have to reach, we use the execution history feature of `angr`. The execution history is an ordered list of events happened and actions performed during mixed concrete and symbolic execution. It acts as a log and records information such as when and which symbolic variables have been created for the state, which addresses have been traversed, what symbolic constraints have been placed on the state, etc. We insert special events into each execution state's execution history to record target control-flow graph nodes, as well as related events such as reaching those target control-flow graph nodes and satisfying the associated predicate of a system call.

4.6 Evaluation

We evaluate our approach on two malware samples. The first sample is an artifical malware sample we developed that is based on the features of the Kaiten, Aes.Ddos and Amnesia families [36, 121]. It consists of 516 lines of C code. It first checks whether it executes in a virtual environment by reading the contents of the `/sys/call/dmi/id/sys_vendor`

file. If the content includes the string `"QEMU"`, it exits. Amnesia samples are known to use this trigger[12], however, instead of exiting, they attempt to wipe the hard drive of the machine.

If no virtual environment is detected, our 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. The implementation of this feature relies on command execution in a shell and the `sysinfo`[13] system call, the latter of which is known to be used by AES.DDOS samples[14]. The results of the `sysinfo` system call are periodically resent to the command and control server, thereby keeping the attacker up-to-date regarding the state of the infected machine.

Our piece of malware 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. We copied the denial-of-service implementation from a publicly available Kaiten sample[15].

The second sample is a slightly modified version of the previously mentioned Kaiten sample. It implements its own IRC protocol parser and expects remote commands to be delivered as IRC private messages. Some commands are used to launch denial-of-service attacks, execute shell commands and download files. Others serve management purposes such as enabling/disabling packet sending, changing IRC servers, etc.We chose this sample because its execution relies heavily on its environment. In order to trigger any malicious behavior, the sample must be able to communicate over the network. It needs to connect to the IRC server at the preprogrammed address and log into the preprogrammed IRC channel. The sample uses randomly generated strings as nick and user name in the IRC communication; the seed is calculated from the system time, the process ID and the parent process ID. Once connection to the IRC channel has been established, the correct IRC private message must be received in order to trigger any behavior implemented in the sample.

Both pieces of malware poses several challenges for analysis. First, due to our assumptions and its implementation, a large number of execution paths are available for analysis, the sources of which are the following:

1. Environmental data: Our piece of malware relies on the system time, process IDs and communication over the network. As we assume no prior knowledge about its functionality, our analysis has to analyze all those inputs using symbolic variables, leading to many branches.

2. String handling: Our piece of malware processes network input using standard libc functions such as `strlen` and `strcasecmp`. These functions typically loop over the

[12] https://unit42.paloaltonetworks.com/unit42-new-iotlinux-malware-targets-dvrs-forms-botnet/ (Last visited: Jun 8, 2020)
[13] https://www.man7.org/linux/man-pages/man2/sysinfo.2.html (Last visited: Jun 8, 2020)
[14] https://blog.syscall.party/post/aes-ddos-analysis-part-1/ (Last visited: Jun 8, 2020)
[15] https://packetstormsecurity.com/files/25575/kaiten.c.html (Last visited: Jun 8, 2020)

string character by character. As their inputs are returned from the kernel, our analysis must consider each of the characters a symbolic variable. Such loops are known to contribute to the path explosion problem.

3. Infinite loop: Our piece 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.

Another challenge is in the malware samples' logic. In case of receiving a command in a network message, they dispatch the message to the appropriate handler function via a jump table. Jump tables are represented in the control-flow graph by nodes with multiple call edges leading to different handler functions. The use of jump tables decreases the accuracy of shortest path calculation: the shortest path is always to take the correct call edge, even if said edge is infeasible.

4.6.1 Setting up the experiment for the artificial malware sample

We set the address of the command and control server to `127.0.0.1` and replaced the `gethostbyname` function[16] with a symbolic summary in order to avoid symbolically analyzing a DNS lookup for the target of the denial-of-service attack (the sample accepts both domain names and IP addresses). Our symbolic summary function returns an unconstrained IPv4 address. Finally, we compiled the source code[17] of our piece of malware for the ARM platform and performed our analysis on the resulting binary.

We set up three target system call patterns of malicious behavior as targets for our analysis: detection of virtual environment, system parameter leak and denial-of-service attack. For each target system call pattern, we save the path conditions of states that have reached all system calls in the target system call pattern and satisfied each associated predicate. Note, that due to the large number of execution paths and the previously mentioned challenges regarding shortest path calculation, there may be multiple states that are analyzed at any given time. We save the path conditions of all states that complete the target system call pattern in the same iteration of analysis.

The target system call pattern describing the detection of a virtualized environment has two system calls with associated predicates. The first system call is `open`, which opens a file at a given location. The associated predicate checks whether the file to be opened is in the folder `/sys/call/dmi/id/`. This folder contains special files detailing many aspects of the analyzer machine. The second system call is `exit_group`, which is a system call used to terminate all threads associated with a process. This system call is also used by libc functions, such as `fopen` and `fgets`, if certain reliability checks fail (e.g., the system runs out of memory) and the process is forced to abort. As reliability conditions are out of scope for our analysis, we did not allow analysis to pursue them.

[16] http://man7.org/linux/man-pages/man3/gethostbyname.3.html (Last visited: Jun 8, 2020)

[17] The source code used in this study is available at https://doi.org/10.5281/zenodo.3903492 under the GNU Affero General Public License version 3.

The associated predicate for `exit_group` checks whether the system call was invoked at the correct location in our piece of malware.

The target system call pattern for system parameter leak consists of three system calls. The sequence starts with `socket` and the associated predicate checks whether the socket to be created is a TCP socket, i.e., the socket for communicating with the command and control server in our piece of malware. The second system call is `sysinfo`, which returns the system parameters of the infected machine. The associated predicate always returns true because we simply require invocation. We note that there may be programs in which system calls `socket` and `sysinfo` are in reverse order. The last system call is `write` which can be used to write data to a file descriptor. If the file descriptor denotes a socket, the system call causes the data to be sent. The associated predicate checks whether the file descriptor is the previously created TCP socket and whether any of the characters to be written is derived from the returned values of `sysinfo`.

The target system call pattern describing a denial-of-service attack consists of two system calls. In order for a malware sample's infected host not to be overloaded with responses to the flood of packets sent during the denial-of-service attack, the source IP address must be spoofed. Such spoofing requires the attacker to specify a specific address in the IP header, which can be achieved using raw sockets. Therefore, the first system call is `socket` and the associated predicate checks whether the socket to be created is a raw socket. The second system call in the target system call pattern is `sendto`, used to send messages via sockets, and the associated predicate checks whether the socket through which the message should be sent is a raw socket.

We ran the analysis 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. We also restricted angr to 330 Gb of memory.

4.6.2 Results on the artificial malware sample

Unfortunately, while generating the control-flow graph with context sensitivity level 1, angr did not flag the IR blocks implementing the jump table as producing unconstrained successors. As a result, the jump table was not treated as a potential extension point, forcing us to specify the missing edges manually.

An execution path that completes the target system call pattern for detecting a virtualized environment was found in ca. 7 minutes (437.17 seconds) and required a total of 5.96 Gb of RAM. The first system call, `open`, is located close to the beginning of our piece of malware's `main` function which allowed shortest distance symbolic execution to quickly find an execution state which satisfies the associated predicate. Invocation of the second system call, `exit_group`, can be found at multiple program points, the closest of which is the one that is necessary to complete the target system call pattern. Therefore, analysis did not need to select different invocation instances of `exit_group` in the target system call pattern. In order to reach the necessary invocation instance, analysis had to recover the conditions checking the characters in the opened file, generating a total of 71 execution paths before an execution path satisfying all predicates in the target system call pattern was found. The path condition of this execution path contains constraints

Table 4.4: Solutions to the symbolic characters that 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`

which show the necessary ASCII values for the characters to be read from the opened file: `"Q"`, `"E"`, `"M"` and `"U"`.

Finding an execution path which completes the target system call pattern for leaking system parameters completed in ca. 9 minutes (568.09 seconds) and used 5.8 Gb of RAM. Finding an execution path that first creates a TCP socket and then calls `sysinfo` were completed with ease. However, the data returned by `sysinfo` is unconstrained in our analysis as it is read from the operating system. As a result, leaking it requires transforming it into a string representation, which in turn requires fixing its value. Thus, mixed concrete and symbolic execution generated many execution paths in order to complete the third phase. There were four execution paths for which shortest distance calculations returned equal values; their analysis completed the target system call pattern in the same iteration.

Our analysis was able to find an execution path, which completes our target system call pattern for denial-of-service attack in ca. 5 hours (299 minutes) and required 41.4 Gb of RAM. The first control-flow graph node for the `socket` system call to be selected did not create a raw socket; therefore, analysis had to restore an execution state at the entry point of our piece of malware. The next selected control-flow graph node was located in the function implementing the denial-of-service attack. In order to reach that invocation instance, analysis had to determine the correct command to be received from the command and control server. Due to our piece of malware's complexity, which includes a jump table and several string handling functions, analysis spawned hundreds of execution paths.

The path conditions of the three execution states which satisfy all predicates in the target system call pattern shed light on the necessary trigger, as shown in Table 4.4. We manually fed the concrete input returned by the solver to our piece of malware and found that it indeed triggers the execution of the denial-of-service attack. Note, that our piece of malware performs input validation before launching the denial-of-service attack, requiring both the command and two parameters, separated by spaces. The first parameter is the target to be attacked, the second is the duration of the attack. In

our experiment, we hooked the `gethostbyname` function with a symbolic summary to avoid symbolic DNS resolutions, therefore, characters 4-7 do not have any constraints associated with them. The second parameter, on the other hand, appears to be missing from the concrete input. This is caused by our piece of malware's logic: it uses the `atol`[18] function to convert the second parameter from its string form to the numerical representation. Behind the second space, the terminating `NULL` is interpreted as the second parameter, for which `atol` returns 0. The 0 is treated as the duration for the attack (0 seconds), causing our piece of malware to send a single packet.

4.6.3 Experiment setup for the real malware sample

Before we applied our prototype implementation to the publicly available Kaiten sample, we made a few modifications to it, which we describe here. First, we downloaded its publicly available source code[19]. Then, we shortened all strings in the jump tables of the source code to contain only a single character and the terminating null. With this modification, we can contain the path explosion of looping over strings to a certain extent. Note, however, that the modified sample still includes multiple jump tables organized into layers with each layer requiring multiple characters with specific values. Therefore, even with this modification, the sample still requires a string with multiple characters to invoke the necessary handler functions. We also set the address of the IRC server to `127.0.0.1` in order to avoid symbolically analyzing a DNS lookup. Finally, we recompiled the modified source code for the ARM platform and performed our analysis on the resulting binary.

As the target behavior, we selected one of the functions launching denial-of-service attacks (`tsunami` in the source code). The attack is executed in a child process and sends spoofed packets to the target IP specified in the command. We inserted a call to the `kill` libc function before the child process is created and set the underlying `kill` system call as our target. Because there is only a single program point of interest to be targeted globally, we only needed shortest distance symbolic execution as the path selection strategy. Note, that the `kill` system call is used in other functions as well, therefore, we only accept reaching it, if it is done via the `tsunami` function.

In order to reach this function, mixed concrete and symbolic execution has to simulate the communication with the IRC server and "send" a specific string to the sample. The string must meet the following requirements:

1. The sample must interpret its first part as an IRC private message, i.e., it must start with the corresponding code from the jump table of IRC message-handling functions (`4` in our case).

2. It must contain the preprogrammed name of the IRC channel to which the sample logged into (`#` in our modification).

[18] http://www.cplusplus.com/reference/cstdlib/atol/ (Last visited: Jun 8, 2020)
[19] https://packetstormsecurity.com/files/25575/kaiten.c.html

Table 4.5: 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

3. It must be intended for the sample, either by specifically mentioning the sample's IRC nick (randomly generated) or by using a wildcard character.

4. The sample must interpret its last part as a command for launching the DoS attack implemented in `tsunami`, i.e., it must contain the corresponding code from the jump table of command-handling functions (`0` in our case).

Unfortunately, while generating the control-flow graph with context sensitivity level 1, `angr` did not flag the IR blocks implementing the jump tables as producing unconstrained successors. As a result, jump tables were not treated as potential extension points, forcing us to specify the missing edges manually.

We ran our analysis on the same machine as before: 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. We restricted angr to run with 100 Gb of memory.

4.6.4 Results on the real malware sample

Table 4.5 shows the performance of our prototype implementation on the modified 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, and

4. other management tasks, e.g., concretizing stack pointers when necessary, logging events, checking if our target was reached, etc.

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

The execution time of simulating execution paths can be accredited to the logic of the sample. During our tests, analysis encountered addresses, whose simulation took hours for mixed concrete and symbolic execution. These addresses were part of libc, including `rand` and multiple string manipulating functions whose simulation involved computations with complex symbolic values. `rand` is used by the modified sample to generate random 1-character-long strings for communication with the IRC server. While the generated string for the nick 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, we replaced `rand` with angr's built-in symbolic summary 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, we did not influence the execution of string manipulations and settled for the increased execution time.

4.7 Discussion

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, our piece of malware sets an upper limit of 4096 on the number of characters it reads from a socket with one call. As our symbolic summary 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. Our piece of malware 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 simulation, 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, our naming convention regarding symbolic variables tells analysts at which system call a variable was introduced to analysis. Thereby analysts gains semantic information that the symbolic variable's value can be interpreted as characters.

4.8 Conclusion

In this chapter, we proposed an approach to determine what inputs must be provided and what environmental conditions must be met in order to trigger undocumented, hidden behaviors in binary programs. Our proposed approach represents hidden, malicious behaviors as a sequence of system calls with associated predicates, which we refer to as the target system call pattern. In order to find execution paths that correspond such behavior, our approach consists of three techniques. First, we model the environment at the operating system level by providing symbolic summary functions of system calls. Our summary functions have the same number and type of arguments as their real-world counterparts, but introduce fresh symbolic variables in order to model the effects of system calls. Second, we use mixed concrete and symbolic execution and prioritize between available execution paths using a multi-level path selection strategy. Our strategy takes into consideration how far execution paths have advanced in the target system call pattern, whether they can reach interesting program points in new contexts, as well as shortest distance symbolic execution. The last prioritization technique relies on a semantically correct, complete inter-procedural control-flow graph, which is often unavailable for binary programs due to indirect jumps. Therefore, our approach is designed to allow for incorrect/missing edges and/or nodes, as well as actively adding missing edges to the control-flow graph based on existing execution states. Third, we deploy a mechanism for automatically finding program points that can advance an execution path's progress in the target system call pattern.

We implemented our approach using angr and evaluated it on two malware samples: an artificially created sample which incorporates ideas from the Kaiten, Aes.Ddos and Amnesia malware families, as well as on a slightly modified version of a publicly available sample from the Kaiten family. The artificial sample checks whether it is executed in a virtualized environment and executes malicious behaviors only if no such sign has been found. It leaks system parameters to the attacker and accepts remote commands to, for example, launch a denial-of-service attack. The slightly modified Kaiten variant implements its own IRC protocol parser and implements various types of denial-of-service attacks which it launches when instructed to do so via the IRC command and control server. Both samples have logic the pose additional challenges because many of their implementation details are known to be hard to analyze symbolically. Nevertheless, our approach successfully found feasible paths for all targeted program points within reasonable time. The path conditions of these paths gave additional insight as to what kind of environment is needed to trigger a specific malicious behavior. However, their manual interpretation can be tedious and further automation would be beneficial. These results have also been published in [C4, C5].

Chapter 5

Proactive security for embedded IoT devices

The threat landscape discussed in Chapter 2 demonstrates 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, as also shown in Chapter 2, there is a wide range of vulnerabilities that needs the be addressed. Doing so individually for each vulnerability is a tedious task, unfortunately, which does not scale well.

Therefore, in this chapter, we turn our 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. More specifically, we design a new mode of operation for embedded IoT devices, which increases redundancy in addition to satisfying security, reachability, liveness, and safety requirements. We call this mode of operation RoViM afúter its core concept: rotating virtual machines. There is a new emerging trend of virtualization with a large impact on the embedded market [104, 78]. We leverage the isolation feature inherent to virtualization to increase the security of embedded IoT devices by periodically restoring them to a compromise-free state. While one virtual machine is being restored, another takes its place to increase availability. We present a prototype implementation, where we realize an IPsec [62] gateway using our RoViM approach. RoViM's design is evaluated with formal verification and the experiments with our prototype implementation shows no significant change to the use-experience.

The chapter is structured as follows. Section 5.1 provides background information on Self-Cleansing Intrusion Tolerance (SCIT), our inspiration for RoViM. The main difference between SCIT and RoViM is that SCIT is a centralized approach, while RoViM is decentralized to increase fault tolerance. Section 5.2 presents the high-level overview of RoViM, the details of which are presented in Section 5.3. Section 5.4 discusses the formal verification of RoViM's design. We present a prototype implementation in Section 5.5 and conduct experiments to measure its performance in Section 5.6. Section 5.7 concludes this chapter.

5.1 Background

RoViM was inspired by the concept of Self-Cleansing Intrusion Tolerance (SCIT) [10]. Instead of using reactive approaches to security, SCIT is a proactive risk management approach that uses virtual machines. A device implementing SCIT enjoys deletion of malware in every minute, restoration to a pristine state, recovery from software deletion attacks and cooperation with reactive approaches to security. SCIT has been implemented in prototypes for several use-cases in the traditional IT environment, including Single Sign On [54] and Service-Oriented Architecture [89].

The SCIT architecture consists of three main components. First, a virtualization platform is needed for the virtual machines but the architecture remains independent of the chosen platform. Second, the SCIT controller is tasked with controlling the rotation of the virtual machines. It is installed on a secure machine within the internal network and acts as a central component. Third, a short-term persistent memory is required for processing data.

To our knowledge, the design principles of SCIT have never been studied in the context of embedded systems. The domain poses interesting challenges for the original concept because devices run smaller applications that are more limited compared to standard PCs and servers. In addition, many devices are not located inside a protected internal network but are deployed out in the field. This renders the usage of a central component inefficient in this context. Therefore, in this chapter, we replace the central controller with a distributed solution. Handling persistent data in SCIT is also a challenge. The difficulty in the original concept arises from the practice of destroying the virtual machine exposed to the network and replacing it with a new one. This process destroys the memory of the virtual machine, therefore, another solution is needed to keep data that need to be persistent. The authors overcame this problem by using a Network Attached Memory, which acts as a shared memory between virtual machines. However, a shared memory can be used as a stepping stone for the attacker from one virtual machine to another. In this chapter, we present another solution to data propagation between virtual machines that enables the close monitoring of persistent data and can be used to efficiently detect possible compromises.

The idea of using multiple virtual machines that provide the same service is also present in the field of state machine replication [108]. However, there is a key difference between state machine replication and RoViM. In the case of state machine replication, multiple virtual machines are used in parallel to process requests, thereby providing fault tolerance to the system. In the case of RoViM, however, there is only one virtual machine that communicates with external entities. Other virtual machines are on cold standby, isolated from the environment, including attackers.

5.2 RoViM's design

Our designed system, RoViM, follows the principles presented by Bangalore and Sood [10] and provides proactive security for embedded devices. However, the design of RoViM

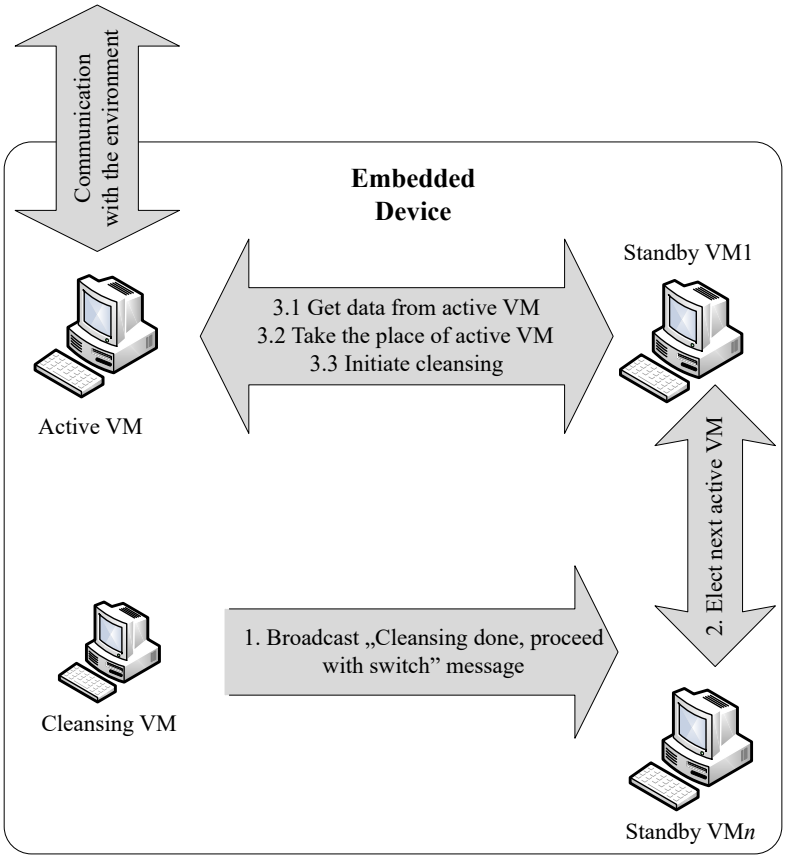

Figure 5.1: High-level overview of RoViM

takes into account not only security, but the potentially high availability requirements of the embedded device as well. The system consists of multiple virtual machines, each of which is capable of performing the same task as the embedded device. The usage of multiple virtual machines provides redundancy and thus contributes to the overall availability of the embedded device.

Before the high-level overview of the system and our assumptions 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.

Our assumptions are as follows. We assume that the virtual machines communicate

via internal communication channels (e.g., virtual LANs), which is not accessible outside of the embedded device. We anticipate communication failures between virtual machines but require the communication channel to notify the system about such a failure. In our threat model, the attacker can interact with the system and compromise the active virtual machine just as he could compromise the embedded device. However, we expect rotations to happen frequently enough that the attacker is unable to compromise standby and cleansing virtual machines via the internal communication channel.

The rotation of virtual machines should be as transparent as possible to outside entities and services. However, devices on the network must know or at least must be notified about the changes in the address of the active virtual machine. Otherwise, packets on the network would not be received by the next active virtual machine. Therefore, we require nodes on the network to accept updates to the network address of the embedded device. For example, in case of Address Resolution Protocol (ARP) on the data link layer, all devices in the local network must process unsolicited ARP replies - which acts as the notification of a rotation - and update their ARP caches.

Figure 5.1 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 [76].

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

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 contain malware, the data on it may become corrupted and malware may be installed. Our designed system can be extended to ensure that no malicious content is propagated to other virtual machines via 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

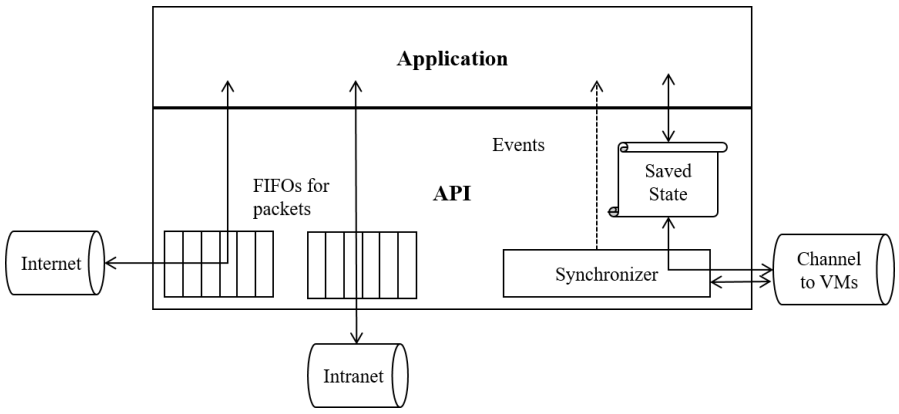

Figure 5.2: Interaction of existing applications and RoViM's API

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 procedure of cleansing must be forced by the next active virtual machine. One such cleansing procedure can be reverting the active virtual machine to a snapshot taken before the deployment of the embedded device.

5.2.1 Stateful applications and RoViM

To complete the three-subphase protocol of replacing the active virtual machine, the running application on the active virtual machine must provide all data necessary for its correct functioning in a serialized form that can be transmitted to other virtual machines. It is also required to be able to restore that data when prompted. As a result, existing applications require some kind of adaptation or extension to work in this paradigm.

We propose making applications aware of the rotation. New applications can be developed with the rotation in mind and existing applications can also be tailored to the rotating environment with little effort from developers. As shown in Figure 5.2, the rotation is implemented by a software layer we refer to as the API. The API should control all resources the application uses to be able to freeze time for the application. In Figure 5.2, FIFO buffers are shown for all network interfaces the application communicates through in order to delay receiving packets. The API communicates with the application via

events to signal different phases of the rotation. When data used by the application is to be copied to the next active virtual machine, the application should provide the data in a serialized form, for example, a file. The data may consists of variables, configuration files, etc. The serialized form is transmitted to the next active virtual machine where the application reads the serialized form and restores the contents. The serialized form of the data used by the application can also be subjected to input validation and be used to efficiently detect compromises. This approach is advantageous from the security point of view as the API does not interact with the possibly compromised memory of the application.

5.3 The three-subphase interaction

As mentioned before, the active and the next active virtual machines must complete a three-subphase interaction before the cleansing procedure. The interaction can be achieved by the following protocol of three subphases.

5.3.1 Subphase 1 - Transferring the application state

During subphase 1, the next active virtual machine receives the data of the application running on the active virtual machine. The subphase starts with a trigger message from the next active virtual machine and acts as a request for the application data in serialized form. The trigger causes the active virtual machine to transmit the requested data to the next active virtual machine. Note that the transmission relies on no specific protocol, the details of transmitting the serialized form is left to the implementation. After receiving the trigger message, the active virtual machine must not process incoming packets. A processed packet at this time could change the application data, rendering the transmitted serialized state out-of-date. Instead, to avoid packet loss, packets are put on hold in buffers until one of the two virtual machines is ready to process them. We note, that a compromised active virtual machine could actively obstruct this subphase by continually refusing to communicate. However, this scenario is out of scope due to our assumption that the attacker cannot use the internal communication channel as an attack interface.

As we anticipate communication failures between virtual machines, virtual machines set a timeout during which messages must arrive. If a message is not received within the time limit, it is considered lost. After suffering a specific amount of lost messages at either virtual machine, that virtual machine assumes the channel to be broken and aborts the protocol without further notice. The time limit and the maximum number of lost messages should be configured with respect to the availability requirements of the embedded device.

5.3.2 Subphase 2 - Configuration of network interfaces

Subphase 2 is attempted only if the application running on the next active virtual machine has the data to be restored. After all, the next active virtual machine has no means

of performing the task of the embedded device, if it is unable to process packets because of the absence of application data. During subphase 2, the active and next active virtual machine configure their network interfaces and notify the networks about the change in the address of the embedded device.

At the beginning of subphase 2, the next active virtual machine tries to bring its interfaces up through which the application expects packets. If the process is successful, the virtual machines continue the protocol; if unsuccessful, the parties need to abort the protocol. In either case, the active virtual machine is notified about the outcome. In case of success, the next active virtual machine notifies the network about the change in the address of the embedded device and instructs the active virtual machine to bring its interfaces down.

Virtual machines might experience communication failures during subphase 2 as well. To recover from losing the message containing the outcome of bringing up the interface of the next active virtual machine, the active virtual machine sets a time limit. When the time limit is exceeded, the active virtual machine must poll the next active virtual machine for the outcome. Why not abort the protocol? Let us assume for a moment that after a specified amount of polling for status, the active virtual machine deems the communication channel broken and aborts the protocol. At this point both virtual machines are capable of processing incoming packets: both have the data for the application, the correct networking configuration and are accepting packets from the outside. Now, we have two virtual machines as the active virtual machine. Depending on the timing of their packets sent, nodes in the network might repeatedly update the address of the embedded device and transmit packets to one of the virtual machines. However, the embedded device and the outside world would lose synchronization, as the virtual machines would update their data based on different packet flows.

5.3.3 Subphase 3 - Optional buffering

During subphase 3, packets buffered during the interaction are relocated to the virtual machine capable of processing them. If subphases 1 and 2 finished successfully, buffered packets are transmitted to the next active virtual machine and processed there. Before relocating the buffered packets, the next active virtual machine requests information about the size of the buffer. Depending on the received size, it decides whether the time necessary for processing the packets is within the availability requirements. If the protocol was aborted, the buffered packets are processed by the active virtual machine. Buffered packets suffer latency, which depends on the size of the application data, the number of packets arriving to the active virtual machine during subphases 1 and 2, and the network throughput between the two virtual machines. Depending on the availability requirements, the introduced latency may or may not be acceptable. In some cases, where packet loss is acceptable (e.g., communication using UDP), buffering packets should be disabled and packets arriving during subphases 1 and 2 should be dropped.

Even though subphase 3 is optional, chance of recovery from possible communication failures is added to the protocol. When the active virtual machine receives the command to bring its interfaces down and subphase 3 is enabled, it sets a timeout during which the

request for information about the buffered packets must arrive. If the request does not arrive, the active virtual machine forcibly sends the information to the next active virtual machine and waits for the decision. If the decision does not arrive in time, it is treated as a refusal. At the next active virtual machine, after the request for information is sent, a timeout is set during which the requested information has to arrive. If it does not arrive, the next active virtual machine can retry and then ultimately abandon subphase 3, deeming minimizing latency more important than avoiding packet loss.

At the end of subphase 3, the active virtual machine must be cleansed. However, it may have been corrupted during its interaction with the environment, therefore, it cannot be trusted to perform its own the cleansing. Therefore, it is the next active virtual machine that sends the notification to the hypervisor to revert the active virtual machine back to a pre-defined snapshot (the known clean state).

5.4 Formal evaluation of RoViM's design

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 protocol is restored in case of errors and the active virtual machine is still in the active role.

It is also important to check whether following the previously proposed protocol can lead to virtual machines having inconsistent states and whether the protocol has any deadlocks. To answer these questions, we subjected the protocol to formal verification using Uppaal [13]. The formal verification was not aimed to finding security issues (these are discussed in Section 5.7), but to check the correctness of the protocol with respect to functionality. The models presented here were also not used as a specification of the protocol and serve only the purpose of verification.

Uppaal is an integrated tool for modeling, verifying, and validating real-time systems using networks of timed automata, extended with data types. The tool has three main components: a description language used to describe system behavior; a simulator, which can validate possible dynamic executions and enables early fault detection; and a model-checker, which explores the state-space of the system, and checks invariant and reachability properties.

To verify our proposed protocol, all participants must be modeled as a network of timed automata. We have three participants: the elected standby virtual machine (called New), the active virtual machine (called Old) and a node on the local network (called Network Node). The latter provides network connectivity to the embedded device and can be, for example, a switch, a router, or a gateway. In Uppaal, participants are called processes. Each process consists of locations (depicted by filled circles) and edges between its locations (depicted by arrows). Edges are annotated with selections, guards, synchronization and updates. The labels have the following meaning:

```
urgent chan ack;              // acknowledgement
urgent chan trigger ;         // trigger state sync in old
urgent chan sync_state;       // sync state with new
urgent chan status_up;        // new's interface is up
urgent chan status_down;      // new's interface is down
urgent chan invalid_arp ;     // invalidating ARP cache
urgent chan bring_down;       // bring down old's interface
chan packet[5];               // transmitting packets:
                              //     0 - to old,
                              //     1 - to new,
                              //     2 - from old,
                              //     3 - from new,
                              //     4 - from old to new
chan status_poll ;            // polling status of new
```

Figure 5.3: Modeling communication channels in Uppaal

- Selection: non-deterministically bind a given identifier to a value in a given range, e.g., `i : int[0,1]` binds either 0 or 1 to the identifier `i`. On the GUI, Selections are shown with the color dark yellow.
- Guards: the edge is enabled in a state if and only if the guard evaluates to true, e.g., `i == 0`. On the GUI, guards are shown with the color green.
- Synchronization: synchronize processes over channels. On the GUI, synchronizations are shown with the color light blue.
- Update: the expression is evaluated and, as a side-effect, the state of the system is updated with the value, e.g., `i += 1`. On the GUI, updates are shown with the color dark blue.

Communication between the participant is modeled with channels which can be defined with the `chan` keyword, as shown in Figure 5.3. Channels can also be defined as arrays. Uppaal also defines urgent channels, and no delay is allowed, when a transition using urgent channels is enabled. In a real environment, this means that if a message can be sent, it will be sent as soon as possible.

5.4.1 Network node

Figure 5.4 shows the model of a node on the network. The node has two locations that correspond to which virtual machine packets are sent: in `ToOld`, it sends packets to the active virtual machine (Old), while in `ToNew`, it send packets to the next active virtual machine (New). Edges between the locations define transition: if the process is a recipient during synchronization, the channel name is followed by a `?`, if the process is the sender, then an `!` is written after the channel name. Changing locations happen only

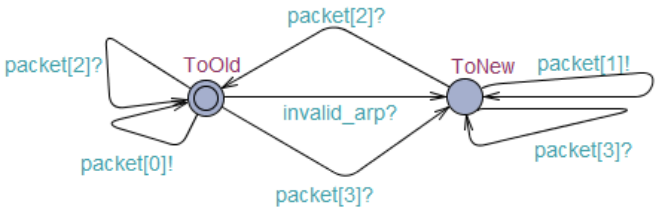

Figure 5.4: Model of a Network Node

```
clock  timer;                  // timer
int [0,1]  hasError  = 0;      // whether an error  happened
int [0,2]  numErrors = 0;      // how many errors happened
int [0,2]  maxErrors = 2;      // maximum number of errors allowed in a location
```

Figure 5.5: Local variables in the model of active virtual machine

if a received packet has another source than the process expects (e.g., packet originates from New, even though the network node would send packet to Old) or the change is explicitly requested by the protocol with a notification about a change in network address.

5.4.2 Active virtual machine (Old)

To create the model of the active virtual machine (Old), local variables are also needed. The local variables are shown in Figure 5.5. To model time and timeouts, the variable `timer` is used, which is of type `clock`. Uppaal uses a dense-time model where clock variables evaluate to real numbers. Clocks in the system progress synchronously. Unfortunately, Uppaal does not model random communication failures, therefore, it has to be introduced to the model manually. For this reason, the variable `hasError` of type `int` is added as a local variable. Unless constraints are placed on a variable regarding its value, Uppaal will verify the model using all possible values for each variable. In case of `int`, it is both time and space consuming, and since `hasError` tells whether a communication error happens, the values 0 and 1 are used as lower and upper constraints respectively. While the variable could be defined as `bool`, it is not possible to select a random boolean value on an edge, which would be needed to model random communication failures. The number of errors (`numErrors`) in a location must also be tracked to know how many retries the system attempted and whether the maximum number of allowed errors (`maxErrors`) has been reached (this causes the protocol to be aborted).

Figure 5.6 shows subphase 1 of the three-subphase interaction from the active virtual machine's point of view. Initially, the active virtual machine is in the `ActAsActive` location, in which it accepts and sends packets whenever needed. When it receives the `trigger` message from the next active virtual machine, it transits to the`InitiateSync` location. While the transition is underway, a random value is selected and is given to

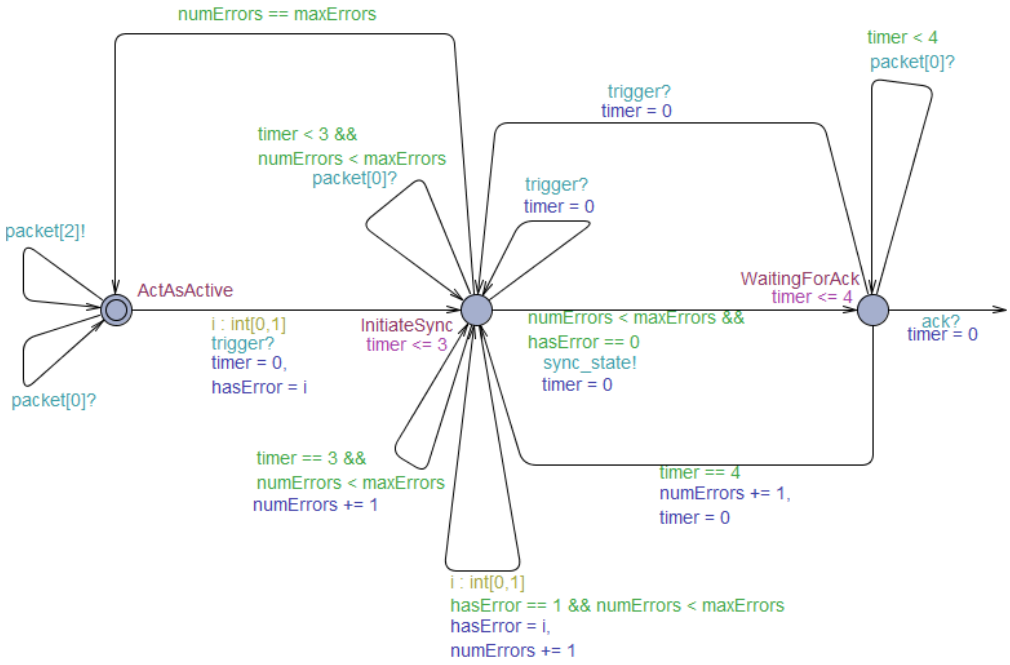

Figure 5.6: Modeling subphase 1 for the active virtual machine

the local variable `hasError`.

In Uppaal, a process is allowed to stay in a location infinitely, unless a clock specifies the maximum amount of time that can pass without using an enabled transition. Therefore, a maximum amount of time is specified until the active virtual machine can try to send the `sync_state` message, as well as for all other locations in the model. The message gets across, if the randomly selected value of `hasError` is 0 and fails if the value is 1, provided that the maximum amount of retries is not exceeded. Each time sending fails, the counter `numErrors` is incremented, until it reaches the maximum amount of allowed errors. If that value is reached, the protocol is aborted by transitioning back to the location `ActAsActive`. Because switching to the next active virtual machine is underway, no packets can be processed, instead, incoming packets may be buffered. Therefore, incoming packets (`packet[0]?`) are accepted in this location (and are buffered) as long as time at the location is within the specified limits and not too many errors happened. Due to timing issues, it is possible to receive additional `trigger` messages in this location. If that happens, the clock is reset and the virtual machine continues to send the `sync_state` message.

When the `sync_state` message gets across to the next active virtual machine, the active virtual machine enters the `WaitingForAck` location. In this location, it waits for the acknowledgement that the next active virtual machine received the application data. If the response does not arrive within the timeout, the active virtual machine goes back to the location `InitiateSync` and retries, meanwhile also incrementing the number of errors that happened. Due to timing issues, it is possible to get a `trigger`

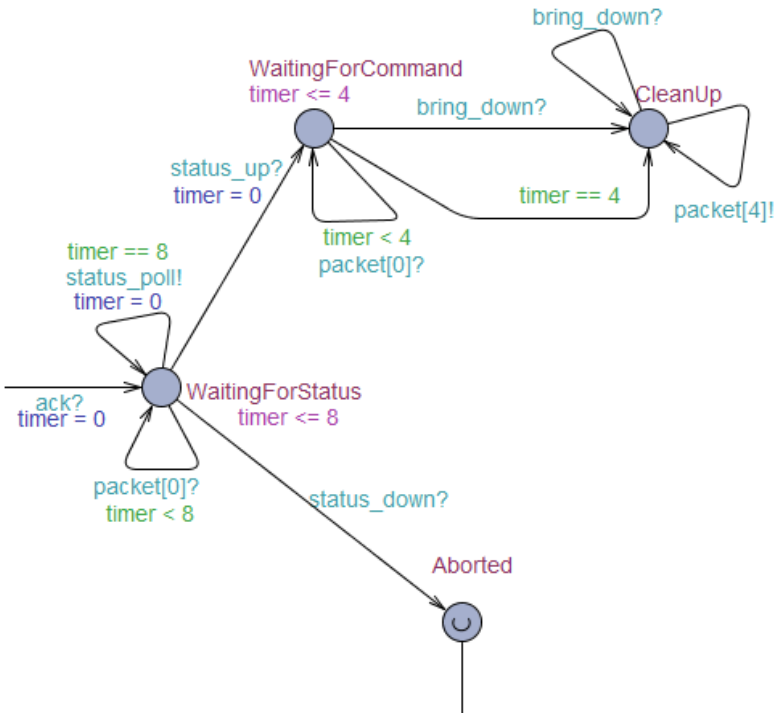

Figure 5.7: Modeling subphases 2 and 3 for the active virtual machine

message in this location. In this case again, the active virtual machine goes back to the location `InitiateSync`. As in the previous locations, incoming packets are accepted within timeout (and are buffered). When the acknowledgement arrives from the next active virtual machine, the active virtual machine enters subphase 2.

For the active virtual machine's point of view, subphase 2 starts when it enters the location `WaitingForStatus`, as shown in Figure 5.7. In this location, the active virtual machine is waiting for the next active virtual machine to notify the active virtual machine about the status its network interfaces. As discussed in Section 5.3.2, if the protocol reaches this point, it cannot be aborted because of timeouts. Instead, when a timeout does occur, the active virtual machine must poll the next active virtual machine for the required information. At this point, network nodes still send packets to the active virtual machine, thus, incoming packets are accepted (and buffered). When the status message of the next active virtual machine arrives, the active virtual machine transitions to the corresponding location.

If the status message from the next active virtual machine is negative (`status_down?`), i.e., the interfaces could not be brought up, the protocol is considered aborted and the active virtual machine transits to the location `Aborted`. In this location, buffered packets are replayed in hopes of preventing packet loss. The location is urgent, i.e., no time is allowed to pass in this location. In real life, as soon as buffered packets are retransmitted, the location is left and the active virtual machine transits to the starting point

```
clock timer;
int [0,1] hasError = 0;          // whether some kind of error happened
int [0,2] maxErrors = 2;         // maximum number of errors allowed
int [0,2] numErrors = 0;         // number of errors happened
int [0,1] interfaceError = 0;    // whether the interface can be brought up or not
```

Figure 5.8: Local variables in the model of the next active virtual machine

`ActAsActive`.

If the status message is positive (`status_up?`), i.e., the interfaces are up and address invalidation will happen, the active virtual machine transits to the state `WaitingForCommand`. Because packets may still be sent to the active virtual machine in this location (it has no knowledge about when the address invalidation happens), packets are accepted (and buffered). The active virtual machine waits for a specified amount of time to get the final `bring down interface` message (`bring_down?`). Irrespective of whether the message arrived within timeout or not, the active virtual machine enters subphase 3 in the location `CleanUp`. Because buffering is optional in the protocol, its messages are not modeled. Instead, the simpler approach is to let the model decide: if buffering is on, the synchronization edge `packet[4]!` is taken by the model, if not, the edge is not enabled. Because the active virtual machine cannot wait for the `bring down interface` message, the message may arrive when the virtual machine is in the location `CleanUp`.

5.4.3 Next active virtual machine (New)

The model of the next active virtual machine also needs local variables, as shown in Figure 5.8. Most of the variables are the same as in the model of the active virtual machine. The only new variable is called `interfaceError` and is of type `int`. It serves a similar function as the variable `hasError` in the sense that it is used to model a random error. However, the error is not in the communication but one that may occur while bringing up the network interfaces of the virtual machine. If bringing the network interfaces up is successful, the value is 0, and if not, it is 1. Similarly to `hasError`, this variable could be defined as a boolean value but because boolean values cannot be randomly selected in Uppaal, it is of type `int` and has a constraint on its value: 0 or 1.

The model of subphase 1 from the next active virtual machine's point of view is shown in Figure 5.9. Initially, the next active virtual machine is in the location `Start` and is waiting for the leader election. Because the problem of leader election has been discussed in literature many times, the leader election itself it not modeled and the process is assumed to have won the election. As a result, it immediately transits to the location `LeaderElectionWon`, while randomly selecting a value for the variables representing communication failure. The location has an upper time limit on how long the next active virtual machine should try to send its message. If sending is unsuccessful (`hasError == 1`), or the time limit is reached (`timer == 2`) and the maximum number of allowed errors is not yet reached, the counter for errors is incremented and the virtual

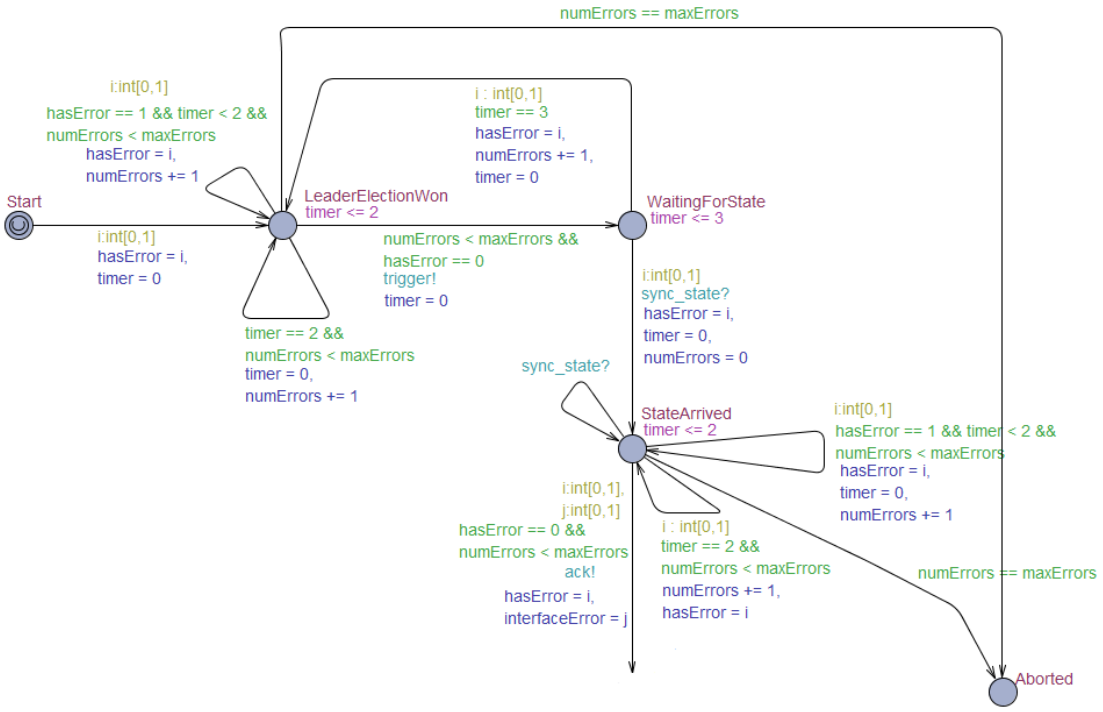

Figure 5.9: Modeling subphase 1 for the next active virtual machine

machine tries again. However, if the maximum number of allowed errors is reached (`numErrors == maxErrors`), the protocol is aborted by transitioning to the location `Aborted`.

When sending the `trigger` message to the active virtual machine, the next active virtual machine waits for the serialized application state in the location `WaitingForState`. If the time limit of waiting is reached, the error is recorded and the virtual machine transitions back to the location `LeaderElectionWon`. When the `sync_state` message from the active virtual machine arrives, the next active virtual machine transitions to the location `StateArrived` and attempts to send the acknowledgement. If sending is not possible (either because of communication failure or the time limit set on this location), the error counter `numErrors` is incremented until the maximum number of allowed errors is reached. If the maximum is reached, the protocol is aborted by transitioning to the location `Aborted`. Due of timing issues, a `sync_state` message may arrive while the virtual machine is in this location (e.g., because of a retry from the active virtual machine). If the acknowledgement can be sent, the virtual machine enters subphase 2 of the protocol.

Figure 5.10 shows the models of subphase 2 and 3 from the next active virtual machine's point of view. When the next active virtual machine enters subphase 2 by transitioning to the location `BringUpNewInterface`, a value is selected randomly for `interfaceError` that indicates whether the interface could be brought up or some sort of

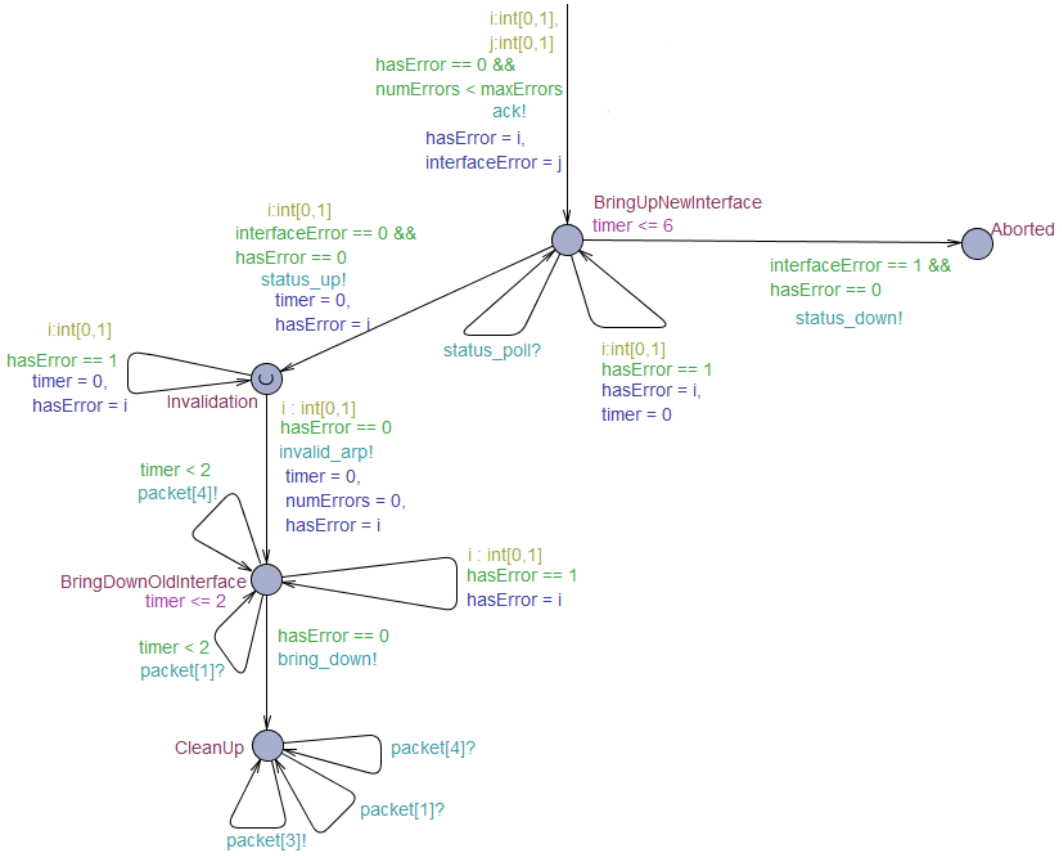

Figure 5.10: Modeling subphases 2 and 3 for the next active virtual machine

error happened. If the interface cannot be brought down and the message `status_down` can be sent, the protocol is aborted. If the interface is up and the message `status_up` can be sent, the virtual machine transitions to location `Invalidation`. If no message can be sent, the virtual machine retries. In this location, a `status_poll` message can also be received, if the status message had not been sent in time.

In the location `Invalidation`, the next active virtual machine tries to send a message to the network node informing it in the change in Layer 2. If sending fails, the virtual machine retries immediately, if it is successful, it transitions to the location `BringDownOldInterface`. Because the interface of the virtual machine is up and the serialized application state is available, the next active virtual machine can process packets. In this location, it tries to send the `bring_down` message to the active virtual machine and moves to subphase 3, if sending succeeds. Because of the model of the active virtual machine, the optional buffering part of the protocol is only modeled by receiving the buffered packets in the location `CleanUp`.

5.4.4 System

```
// Place template instantiations here.
Old = OldActive();
New = NewActive();
R = Router();

// List one or more processes to be composed into a system.
system Old, New, R;
```

Figure 5.11: RoViM modelled as a system of timed automata

With the models ready, the system can be composed using the previously discussed timed automata. First, for each model, an instance is created. Then, the system is defined as the instances. The source code to define the system of timed automata is shown in Figure 5.11.

5.4.5 Checking properties of the system

With the system ready, it is uploaded to the model-checker to test whether it satisfies the requirements. The model-checker does not evaluate the behavior of the system but instead its state-space. The state-space can be represented with a graph in which every node contains a possible set of states of the system and directed edges are possible changes is the state of the system. Queries to the model-checker are expressed using a simplified version of Timed Computation Tree Logic. The query language consists of path formalae and state formulae. A state formula is an expression, which is evaluated by looking at the state-space. For example, the expression `i == 0` evaluates to true in all states in the state-space where in the system `i = 0`. The state of processes can also be expressed with state formulae by using the syntax of `ProcessName.LocationName`. In Uppaal, deadlock is expressed with a special state formula called `deadlock` which is satisfied for all deadlock states.

Uppaal can be used to check three kinds of properties: reachability, safety and liveness. Reachability properties are satisfied when a path exists from the initial state, such that the state formula ϕ is satisfied by any state along that path. Reachability properties are expressed using `E<>` ϕ. Safety properties are invariants, which are always true in the system. If ϕ should be true in all reachable state, then the path formula is `A[]` ϕ. The path formula `E[]` ϕ says that there should exist a maximal path such that ϕ is always true. Liveness properties express that something will eventually happen, and are expressed using the path formula `A<>` ϕ and ϕ `-->` ψ.

The protocol must have the following properties. First, the reachability property requires that it should be possible for the protocol to end with the virtual machines in consistent state. Two such ends exist: 1) both the active and the elected standby virtual machines aborted the protocol and packets are sent to the active virtual machine, and 2) both the active and the next active virtual machines entered subphase 3 and packets are sent to the next active virtual machine. The first requirement is formulated as

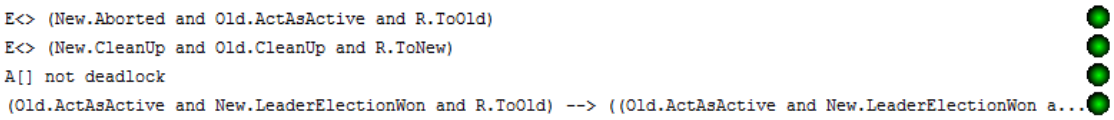

Figure 5.12: Results of formal verification

```
E<> (New.Aborted and Old.ActAsActive and R.ToOld),
```

while the second is

```
E<> (New.CleanUp and Old.CleanUp and R.ToNew)}.
```

Second, there is a safety property the protcol has to conform with: there must be no deadlock in the model. There must not be a state in which the system is unable to transition to another state. This is formulated as

```
A[] not deadlock.
```

And third, the protocol also has a liveness property: from the moment the protocol is started, the protocol must either be aborted and the starting state must be reached, or it must reach subphase 3 eventually. This can be formulated as

```
(Old.ActAsActive and New.LeaderElectionWon and R.ToOld) -->
    ((Old.ActAsActive and New.LeaderElectionWon and R.ToOld) or
    (Old.CleanUp and New.CleanUp and R.ToNew)).
```

Figure 5.12 shows the result of the formal verification. The green lights next to the requirements show that each requirement is met, the model, and our proposed protocol for the three-subphase interaction, has all the previously mentioned properties.

Additional liveness properties may be defined, for example, if the next active virtual machine reaches subphase 3, eventually the active virtual machine must reach it too and the network node must send packet to the new active virtual machine. The formula would be `New.CleanUp --> (Old.CleanUp and R.ToNew)` Or, if the active virtual machine aborts the protocol, eventually the next active virtual machine must abort it as well and packets must be sent to the active virtual machine. This property would be formulated as `Old.Aborted --> (New.Aborted and R.ToOld)`. Both example properties are also successfully verified by Uppaal.

5.5 Prototype implementation

We implemented a prototype of RoViM in the environment shown in Figure 5.13. The prototype implementation is a RoViM-enabled IPsec gateway consisting of four virtual machines (`IPsecGatewayServer`). IPsec [62] is a set of security services for traffic at the IP layer, for both IPv4 and IPv6. These services include access control, connectionless integrity, data origin authentication, protection against replays, and limited traffic flow

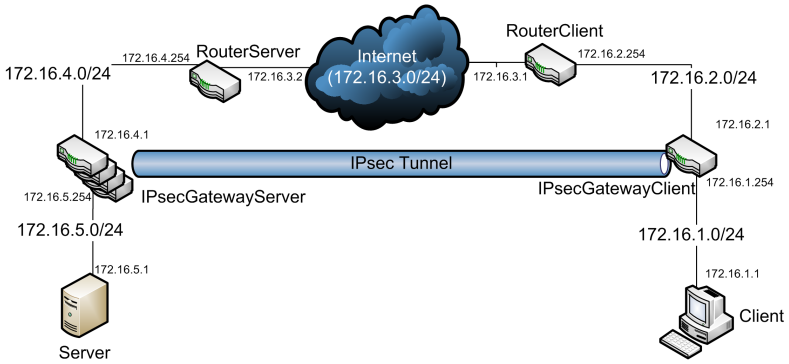

Figure 5.13: Environment for the RoViM-capable IPsec gateway prototype

confidentiality. IPsec creates a boundary between protected and unprotected interfaces (e.g., a host or a network). Traffic traversing through the boundary is subjected to access control, which indicate whether packets should be allowed to traverse with or without protection, or should be discarded.

IPsec relies heavily on the concept of Security Associations (SAs). An SA is a simplex connection that provides security services to the traffic carried. To secure a typical, bi-directional communication channel between to IPsec-enabled systems, a pair of SAs is needed. SAs van be created automatically using the Internet Key Exchange (IKE) protocol or manually by the system administrator.

While an SA is a management construct used to enforce security policy for traffic, the policies that specify what services are to be offered to IP packets and in what fashion, are the Security Policies stored in the Security Policy Database (SPD). When processing a packet, the SPD must be consulted and it must provide three choices for traffic: discard, bypass or protect. If the choice is discard, the traffic is not allowed to traverse the IPsec boundary in the specified direction. In case of bypass, the traffic is allowed to traverse the boundary, but no protection is provided. In case of protect, the traffic is afforded IPsec protection and the SPD must specify the security protocols to be employed, including their mode, security service options, and the necessary cryptographic algorithms.

The RoViM-enabled IPsec gateway prototype in our environment is an end-point of an IPsec tunnel which establishes secure communication between two end-points, the `Client` and the `Server`. Both the `Client` and the `Server` are implemented as virtual machines, and the `Server` implements a file server accessible via HTTP. The Internet itself is modeled as a network between two routers.

In our prototype implementation of `IPsecGatewayServer`, all virtual machines share the same IP address to the outside world, which makes rotations transparent in Layer 3 and above. However, separate addresses are used in Layer 2 to differentiate between the virtual machines. As a result, the destination address of the frame decides which virtual machine receives the frame. It must be mentioned that this solution works only if the Layer 2 address related to an IP address can be forcibly updated at nodes on the local networks. Because we use ARP in the data link layer, notifications about the rotation

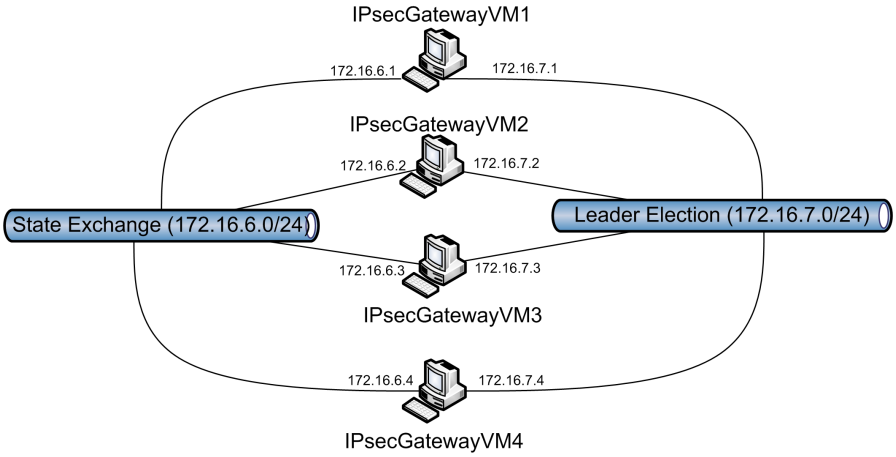

Figure 5.14: Internal networks for managing rotation

are unsolicited ARP replies sent by the gateway.

The rotating virtual machines use two additional networks for internal communication, as shown in Figure 5.14. The `Leader Election` network is used by the standby virtual machines to decide the next active virtual machine. The three-subphase interaction between the active and next active virtual machines happens in the `State Exchange` network. The separation of internal networks ensures that standby virtual machines are separated from the exposed active virtual machine. Virtual machines keep their unnecessary network interfaces down.

Our prototype implementation uses VMware ESXi[1] as the virtualization platform. The virtual machines all run the same operating system, Ubuntu 14.04 Server LTS. Additional packages are installed for the implementation: openssh-server for transferring the application state, ulogd2-pcap for implementing the FIFO buffers, arping for sending unsolicited ARP replies, and python-pip, because the implementation was written in Python.

5.6 Performance evaluation of prototype

We evaluate the performance of the IPsec gateway prototype with respect to packet loss and user experience. We measure the extent of packet loss with the optional buffer feature of subphase 3 enabled in Section 5.6.1 with a stream of ICMP requests and replies. We also measure user experience with a continuous packet flow generated while downloading a large file from the `Server` to the `Client`.

[1] https://www.vmware.com/products/esxi-and-esx.html (Last visited: Jan 5, 2021)

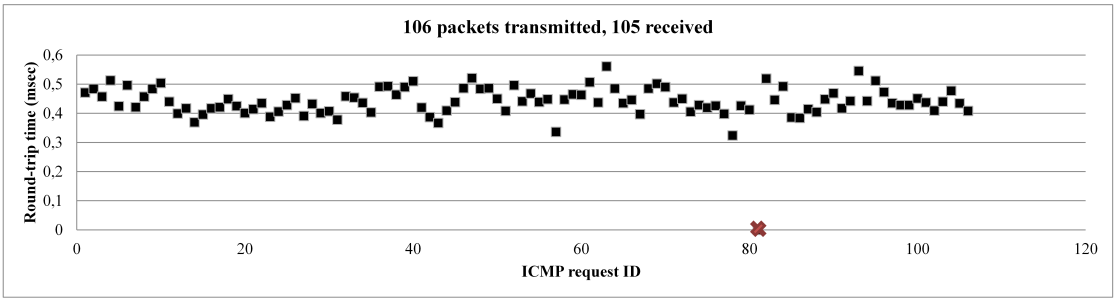

Figure 5.15: Round-trip times of individual ICMP requests and corresponding replies

5.6.1 Measuring packet loss with ICMP packets

The first experiment is aimed at determining whether packet loss still occurs with the optional buffering feature of subphase 3 enabled in the prototype implementation. The packet flow necessary for the experiment must not have any mechanism built in to protect against packet loss. Therefore, we chose ICMP requests and replies with which to perform the experiment. We executed a `ping` command on the `Client` aimed at the `Server`, thus sending ICMP requests and replies between them. Meanwhile, the virtual machines in the IPsec gateway performed one cycle of rotation triggered manually.

The results of this experiment are shown in Figure 5.15. 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. After the Security Associations and the Security Policies of the IPsec tunnel had been transmitted to the next active virtual machine, the active virtual machine and the next active virtual machine entered subphase 3. When the next active virtual machine requested information about the buffered packets, ICMP request #81 had not yet been processed by ulogd2 and an empty pcap file was transmitted to the next active virtual machine. It was only after subphase 3 that ICMP request #81 was processed at the active virtual machine and the request appeared in the pcap file. Running the test multiple times resulted in the same outcome.

5.6.2 Measuring user experience using continuous TCP packet flow

The second experiment we conducted was aimed at finding changes in the user-experience. Therefore, we downloaded a 500 MB file from the `Server` to the `Client`. As in the previous test, the rotation at the RoViM-enabled IPsec gateway prototype was triggered manually. The HTTP protocol used to download the file uses TCP in Layer 4, therefore, the packet flow also provided insight into how the latency introduced by the three-phase interaction influences TCP. We ran Wireshark at the `Server` to observe the packet flow and gather statistics.

The results are shown in Table 5.1. The TCP connection between the `Client` and the

Table 5.1: 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

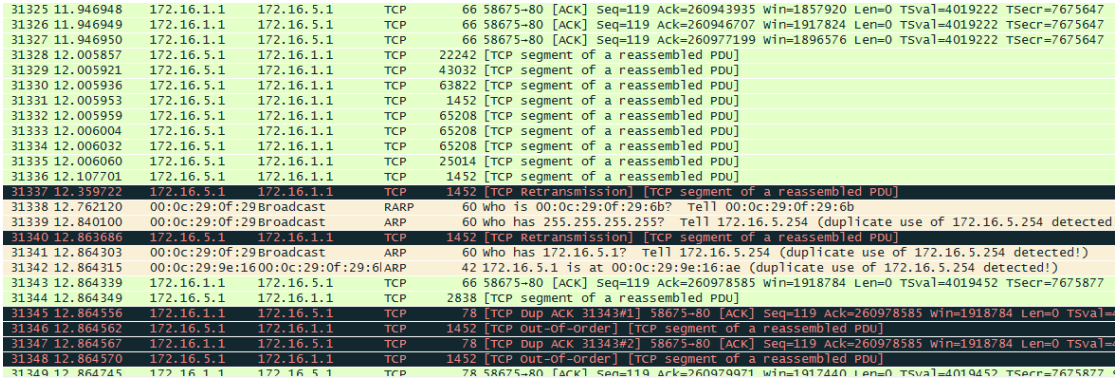

Figure 5.16: Wireshark's "Duplicate IP address configured" warning

`Server` did not break during the rotation. As expected, the rotation introduced latency to the transmission, but the transmission time increased with only 0.004 s, which does not influence the user-experience.

During subphase 2 of the three-subphase interaction, the next active virtual machine brought up its network interface to the server. This 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, once when the interface was brought up, and once when the invalidation of the ARP caches in the 172.16.5.0/24 network happened.

The rotation introduced a significant increase in retransmissions for TCP. To understand the issue here, the retransmission mechanism of TCP must be discussed first. For each segment sent, parties communicating via TCP expect an acknowledgement. If this acknowledgment does not arrive within the retransmission timeout, the segment is sent again. All TCP implementations must use two specific algorithms for computing the retransmission timeout as dictated by RFC 1122 [15]. The algorithms combined adjust the retransmission timeout to the capabilities of the connection link: connections with higher throughput have lower retransmission timeouts and connections with lower throughput have higher timeouts. In our case, up until the optional buffering of subphase 3, the network throughput in the test environment is very high. This is 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. Each

packet in the virtual environment is passed from one port to another on the host. The retransmission timeout calculated by Wireshark was 0.25 s before the rotation. Then, the three-subphase interaction occurs and suddenly, all incoming packets are buffered at the active virtual machine. The exact start of the rotation cannot be seen in the packet flow as the protocol was designed to be transparent in Layer 3 and above. However, acknowledgement for packets 31328 to 31336 on Figure 5.16 does not arrive within timeout, therefore, the `Server` retransmits packets 31328 to 31337, up until the next active virtual machine sends the unsolicited ARP reply (packet 31338). The artificial latency introduced by the rotation makes the TCP implementation of the `Server` believe that some kind of network error has occurred and all segments from packet 31328 to 31336 need to be retransmitted in smaller segments. Retransmissions took 6.556 ms to complete.

The increase in out-of-order segments is also caused by subphase 3 in which previously buffered packets are retransmitted by the next active virtual machine. As discussed in Section 5.3.3, when the next active virtual machine has its interfaces up, new packets are allowed to flow and the buffered ones are retransmitted. In case of this experiment, this results in buffered acknowledgements for previous segments to be transmitted to the `Server` while it is retransmitting, causing the segment flow to become out-of-order.

Based on the discussed issues, it seems that the optional buffering of subphase 3 only hinder the performance of TCP. The experiment was repeated with subphase 3 disabled and as a result, TCP needed only 5.113 ms to retransmit the buffered segments.

5.7 Conclusion

In this chapter, we introduced a new mode of operation for embedded IoT devices, which we call RoViM. RoViM allows devices to cleanse themselves of compromises periodically. The design of RoViM uses multiple virtual machines following the emerging trend of virtualization in the field of embedded devices. We formally verified RoViM's design to prove reachability, liveness, and safety properties. We also developed a prototype implementation of a RoViM-enabled IPsec gateway and showed that the latency introduced by the periodic cleansing does not have a significant impact on user experience. The results presented in this chapter were published in [C6].

The threat model used for the design of RoViM assumes that the attacker does not have enough time between rotations to use the internal communication channel between the virtual machines to compromise the next active virtual machine. However, there may be scenarios in which this assumption is broken. For example, the attacker could compromise the next active virtual machine via the serialized application data transmitted during subphase 1: the attacker could provide malicious content to exploit a vulnerability in the application. The attacker can also send bogus data or he may outright refuse to transmit any data at all. In these scenarios, the application is cut from the data necessary to seamlessly execute the rotation and may become out-of-sync with the outside world. While malicious content or bogus data can be detected with extensive input validation, the denial of service situation arising from the missing data

is not easily handled.

RoViM could also benefit from fault detection. In the current design, a fault in the active virtual machine can render the system useless because the application state is lost. What is more, the fault cannot even be detected before the start of subphase 1, because there is no interaction between the next active virtual machine and the active virtual machines before that point in time. With fault detection, standby virtual machines could monitor the active virtual machine and determine when it experiences faults. They could also save the current application state, if no fault is detected. Unfortunately, this would increase the attack surface of standby virtual machines, therefore, further research is necessary to fully understand the advantages and disadvantages.

The security of RoViM could be further enhanced with a diverse pool of virtual machines. If all virtual machines in the system have the same configuration, the attacker can easily compromise each rotated active virtual machine. In order to mitigate this issue, two countermeasures could be deployed. First, the operating system of virtual machines can be hardened using state-of-the-art techniques, e.g., address space layout randomization (both for the kernel and for user-space applications) and the use of NX bit. Second, if the application running on the embedded device can be ported to multiple operating systems, virtual machines could have different operating systems to further increase diversity.

At the time of writing, the prototype implementation runs in a virtualized environment. As the next step, the code should be ported to an embedded Linux operating system running on a multi-core architecture. Experiments with a real RoViM-enabled embedded IoT device would allow us to further investigate the impact RoViM has on user experience.

Chapter 6

Summary of new results

Embedded IoT devices are special-purpose devices with Internet connection. Internet access provides a number of interesting and innovative application domains to these devices in smart homes and cities, transportation, and healthcare. However, Internet access is also a new attack surface, which needs adequate protection. I made the following contributions in this dissertation to this field.

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][1] and highlighted several important aspects of the security of embedded devices.

THESIS 2.1: I studied the possibility of clustering malware samples based on their binary similarity hashes, specifically, their TLSH similarity score in [C3][2]. I showed that existing clustering algorithms, k-medoid and OPTICS, have unacceptable performance on a large corpus of IoT malware.

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.

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

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

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][3]. 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.

THESIS 3.2: In order to tackle the path explosion problem, I proposed a new analysis method relying on directed symbolic execution to guide analysis towards selected program points, for example, potentially malicious behaviors in binaries. I implemented the proposed analysis method in angr and evaluated it on both artificial and real malware samples in [C5][4]. Our 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 our approach.

THESIS 4: I designed a new mode of operation, RoViM, for embedded IoT devices in [C6][5], which allows them to periodically restore a compromise-free state. I formally verified the design using Uppaal and proved reachability, liveness, and safety properties. I also implemented a prototype of RoViM as an IPsec gateway and also measured its performance. Our measurements showed that the proposed new mode of operation does not have a significant impact on user experience.

[3]Thorsten Tarrach supervised my work for AIT Austrian Institute of Technology GmbH.
[4]Thorsten Tarrach supervised my work for AIT Austrian Institute of Technology GmbH.
[5]Zhendong Ma supervised my work for AIT Austrian Institute of Technology GmbH.

List of own 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).

Bibliography

[1] Alaeiyan, M., Parsa, S., and Conti, M. Analysis and classification of context-based malware behavior. *Computer Communications 136* (2019), 76 – 90.

[2] Anand, S., Godefroid, P., and Tillmann, N. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), C. R. Ramakrishnan and J. Rehof, Eds., Springer Berlin Heidelberg, pp. 367–381.

[3] Anderson, H. S., and Roth, P. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints* (Apr. 2018).

[4] 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.

[5] 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.

[6] Apa, L., and Penagos, C. M. Compromising Industrial Facilities from 40 Miles Away . BlackHat.

[7] Babić, D., Martignoni, L., McCamant, S., and Song, D. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 12–22.

[8] Bai, H., Hu, C., Jing, X., Li, N., and Wang, X. Approach for malware identification using dynamic behaviour and outcome triggering. *IET Information Security 8*, 2 (March 2014), 140 –151.

[9] 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).

[10] Bangalore, A., and Sood, A. Securing web servers using self cleansing intrusion tolerance (scit). In *Dependability, 2009. DEPEND '09. Second International Conference on* (June 2009), pp. 60–65.

[11] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)* (2009), vol. 9, Citeseer, pp. 8–11.

[12] Bayer, U., Kirda, E., and Kruegel, C. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), SAC '10, Association for Computing Machinery, p. 1871–1878.

[13] 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.

[14] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, Association for Computing Machinery, p. 2329–2344.

[15] Braden, R. Requirements for internet hosts – communication layers. https://tools.ietf.org/html/rfc1122, October 1989.

[16] 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.

[17] Burnim, J., and Sen, K. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on* (2008), IEEE, pp. 443–446.

[18] 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.

[19] 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.

[20] 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.

[21] Caselden, D., Bazhanyuk, A., Payer, M., McCamant, S., and Song, D. Hi-cfg: Construction by binary analysis and application to attack polymorphism. In *Computer Security – ESORICS 2013* (Berlin, Heidelberg, 2013), J. Crampton, S. Jajodia, and K. Mayes, Eds., Springer Berlin Heidelberg, pp. 164–181.

[22] Ceron, J. M., Margi, C. B., and Granville, L. Z. Mars: An sdn-based malware analysis solution. In *2016 IEEE Symposium on Computers and Communication (ISCC)* (June 2016), pp. 525–530.

[23] Ceron, J. M., Margi, C. B., and Granville, L. Z. Mars: From traffic containment to network reconfiguration in malware-analysis systems. *Computer Networks 129* (2017), 261 – 272.

[24] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 380–394.

[25] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 6–6.

[26] Chen, N., and Kim, S. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering 41*, 2 (2015), 198–220.

[27] Chipounov, V., Kuznetsov, V., and Candea, G. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices 46*, 3 (2011), 265–278.

[28] Clarke, J. W. RuggedCom - Backdoor Accounts in my SCADA network? You don't say..., April 2012.

[29] Costin, A., and Francillon, A. Ghost in the air(traffic): On insecurity of ads-b protocol and practical attacks on ads-b devices.

[30] Costin, A., and Francillon, A. Short paper: A dangerous 'pyrotechnic composition': Fireworks, embedded wireless and insecurity-by-design. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks* (New York, NY, USA, 2014), WiSec '14, ACM, pp. 57–62.

[31] 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.

[32] Cozzi, E., Graziano, M., Fratantonio, Y., and Balzarotti, D. Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 161–175.

[33] 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.

[34] Cui, A., Costello, M., and Stolfo, S. J. When firmware modifications attack: A case study of embedded exploitation. In *Proceedings of NDSS Symposium 2013* (2013).

[35] Cui, W., Peinado, M., Cha, S. K., Fratantonio, Y., and Kemerlis, V. P. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), pp. 820–831.

[36] De Donno, M., Dragoni, N., Giaretta, A., and Spognardi, A. Analysis of ddos-capable iot malwares. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)* (2017), IEEE, pp. 807–816.

[37] Deng, Z., Xu, D., Zhang, X., and Jiang, X. Introlib: Efficient and transparent library call introspection for malware forensics. *Digital Investigation 9* (2012), S13 – S23. The Proceedings of the Twelfth Annual DFRWS Conference.

[38] Dessiatnikoff, A., Deswarte, Y., Alata, E., and Nicomette, V. Potential attacks on onboard aerospace systems. *IEEE Security and Privacy 10*, 4 (July 2012), 71–74.

[39] European Union Agency for Network and Information Security (ENISA). Existing indicent taxonomies.

[40] Fazzini, M., Prammer, M., d'Amorim, M., and Orso, A. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2018), ISSTA 2018, Association for Computing Machinery, p. 141–152.

[41] Fleck, D., Tokhtabayev, A., Alarif, A., Stavrou, A., and Nykodym, T. Pytrigger: A system to trigger extract user-activated malware behavior. In *2013 International Conference on Availability, Reliability and Security* (Sep. 2013), pp. 92–101.

[42] Forner, E., and Meixell, B. *Out of Control: SCADA Device Exploitation.* Cimation, 2013.

[43] 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.

[44] Geovedi, J., and Irayndi, R. Hacking a bird in the sky: Exploiting satellite trust relationship.

[45] 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.

[46] Godefroid, P. Compositional dynamic test generation. *SIGPLAN Not. 42*, 1 (Jan. 2007), 47–54.

[47] Godefroid, P., Klarlund, N., and Sen, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 213–223.

[48] Gomez, M., Rouvoy, R., Adams, B., and Seinturier, L. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (2016), pp. 88–99.

[49] Hanna, S., Rolles, R., Molina-Markham, A., Poosankam, P., Fu, K., and Song, D. Take two software updates and see me in the morning: The case for software security evaluations of medical devices. In *Proceedings of the 2Nd USENIX Conference on Health Security and Privacy* (Berkeley, CA, USA, 2011), HealthSec'11, USENIX Association, pp. 6–6.

[50] Hansman, S., and Hunt, R. A taxonomy of network and computer attacks. *Computers and Security 24*, 1 (2005), 31 – 43.

[51] Hernandez, G., Arias, O., Buentello, D., and Jin, Y. Smart nest thermostat: A smart syp in your home. Black Hat.

[52] Hu, X., Bhatkar, S., Griffin, K., and Shin, K. G. Mutantx-s: Scalable malware clustering based on static features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (USA, 2013), USENIX ATC'13, USENIX Association, p. 187–198.

[53] Hu, X., Chiueh, T.-c., and Shin, K. G. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, Association for Computing Machinery, p. 611–620.

[54] Huang, F., Wang, C.-x., and Long, J. Design and implementation of single sign on system with cluster cas for public service platform of science and technology evaluation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on* (2011), IEEE, pp. 732–737.

[55] Jang, J., Brumley, D., and Venkataraman, S. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, Association for Computing Machinery, p. 309–320.

[56] Jin, W., and Orso, A. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), pp. 474–484.

[57] Jin, W., and Orso, A. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2013), ISSTA 2013, Association for Computing Machinery, p. 213–223.

[58] Joo, J.-U., Shin, I., and Kim, M. Efficient methods to trigger adversarial behaviors from malware during virtual execution in sandbox. *International Journal of Security and Its Applications 9* (01 2015), 369–376.

[59] Kang, B., Yang, J., So, J., and Kim, C. Y. Detecting trigger-based behaviors in botnet malware. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems* (New York, NY, USA, 2015), RACS, Association for Computing Machinery, p. 274–279.

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

[61] Keefe, M. Timeline: Critical infrastructure attacks increase steadily in past decade, 2012.

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

[63] Kifetew, F. M., Jin, W., Tiella, R., Orso, A., and Tonella, P. Reproducing field failures for programs with complex grammar-based input. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation* (2014), pp. 163–172.

[64] Kinable, J., and Kostakis, O. Malware classification based on call graph clustering. *Journal in computer virology 7*, 4 (2011), 233–245.

[65] 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.

[66] Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., and Eckert, C. Empowering convolutional networks for malware classification and analysis. In *2017 International Joint Conference on Neural Networks (IJCNN)* (2017), pp. 3838–3845.

[67] Kong, D., and Yan, G. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2013), KDD '13, Association for Computing Machinery, p. 1357–1365.

[68] Kornblum, J. Identifying almost identical files using context triggered piecewise hashing. *Digit. Investig. 3* (Sept. 2006), 91–97.

[69] Lattner, C. Llvm and clang: Next generation compiler technology. In *The BSD Conference* (2008), pp. 1–2.

[70] Lee, S., Jung, W., Kim, S., Lee, J., and Kim, J.-S. Dexofuzzy: Android malware similarity clustering method using opcode sequence. In *VirusBulletin* (2019).

[71] 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.

[72] Li, Q., Zhang, Y., Su, L., Wu, Y., Ma, X., and Yang, Z. An improved method to unveil malware's hidden behavior. In *Information Security and Cryptology* (Cham, 2018), X. Chen, D. Lin, and M. Yung, Eds., Springer International Publishing, pp. 362–382.

[73] Li, Y., Su, Z., Wang, L., and Li, X. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2013), OOPSLA '13, ACM, pp. 19–32.

[74] Lindorfer, M., Kolbitsch, C., and Milani Comparetti, P. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2011), R. Sommer, D. Balzarotti, and G. Maier, Eds., Springer Berlin Heidelberg, pp. 338–357.

[75] Lyda, R., and Hamrock, J. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy 5*, 2 (2007), 40–45.

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

[77] 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.

[78] Main, C. Virtualization on multicore for industrial real-time operating systems [from mind to market]. *Industrial Electronics Magazine, IEEE 4*, 3 (September 2010), 4–6.

[79] 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.

[80] Mohaisen, A., Alrawi, O., and Mohaisen, M. Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security 52* (2015), 251 – 266.

[81] Mohaisen, A., West, A. G., Mankin, A., and Alrawi, O. Chatter: Classifying malware families using system event ordering. In *2014 IEEE Conference on Communications and Network Security* (2014), pp. 283–291.

[82] Moser, A., Kruegel, C., and Kirda, E. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)* (May 2007), pp. 231–245.

[83] Naik, N., Jenkins, P., Gillett, J., Mouratidis, H., Naik, K., and Song, J. Lockout-tagout ransomware: A detection method for ransomware using fuzzy hashing and clustering. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)* (2019), pp. 641–648.

[84] Nari, S., and Ghorbani, A. A. Automated malware classification based on network behavior. In *2013 International Conference on Computing, Networking and Communications (ICNC)* (2013), pp. 642–647.

[85] Nataraj, L., Yegneswaran, V., Porras, P., and Zhang, J. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence* (New York, NY, USA, 2011), AISec '11, Association for Computing Machinery, p. 21–30.

[86] Nayrolles, M., Hamou-Lhadj, A., Tahar, S., and Larsson, A. Jcharming: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015), pp. 101–110.

[87] Nayrolles, M., Hamou-Lhadj, A., Tahar, S., and Larsson, A. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process 29*, 3 (2017), e1789. e1789 JSME-15-0137.R1.

[88] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[89] Nguyen, Q. L., and Sood, A. Improving resilience of soa services along space-time dimensions. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on* (2012), IEEE, pp. 1–6.

[90] Nielsen, T. L., Abildskov, J., Harper, P. M., Papaeconomou, I., and Gani, R. The capec database. *Journal of Chemical & Engineering Data 46*, 5 (2001), 1041–1044.

[91] 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.

[92] Pagani, F., Dell'Amico, M., and Balzarotti, D. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (2018), ACM, pp. 354–365.

[93] Pai, S., Di Troia, F., Visaggio, C. A., Austin, T. H., and Stamp, M. Clustering for malware classification. *Journal of Computer Virology and Hacking Techniques 13*, 2 (2017), 95–107.

[94] Park, Y., Reeves, D., Mulukutla, V., and Sundaravel, B. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research* (New York, NY, USA, 2010), CSIIRW '10, Association for Computing Machinery.

[95] Park, Y., Reeves, D. S., and Stamp, M. Deriving common malware behavior through graph clustering. *Computers & Security 39* (2013), 419 – 430.

[96] Parvez, R., Ward, P. A. S., and Ganesh, V. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering* (Riverton, NJ, USA, 2016), CASCON '16, IBM Corp., pp. 116–127.

[97] Peng, F., Deng, Z., Zhang, X., Xu, D., Lin, Z., and Su, Z. X-force: Force-executing binary programs for security applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (USA, 2014), SEC'14, USENIX Association, p. 829–844.

[98] Perdisci, R., Ariu, D., and Giacinto, G. Scalable fine-grained behavioral clustering of http-based malware. *Computer Networks 57*, 2 (2013), 487 – 500. Botnet Activity: Analysis, Detection and Shutdown.

[99] Perdisci, R., Lee, W., and Feamster, N. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (USA, 2010), NSDI'10, USENIX Association, p. 26.

[100] Pham, V., Ng, W. B., Rubinov, K., and Roychoudhury, A. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 1, pp. 891–901.

[101] Qiao, Y., Yang, Y., Ji, L., and He, J. Analyzing malware by abstracting the frequent itemsets in api call sequences. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (2013), pp. 265–270.

[102] Rafique, M. Z., and Caballero, J. Firma: Malware clustering and network signature generation with mixed network behaviors. In *Research in Attacks, Intrusions, and Defenses* (Berlin, Heidelberg, 2013), S. J. Stolfo, A. Stavrou, and C. V. Wright, Eds., Springer Berlin Heidelberg, pp. 144–163.

[103] Rieck, K., Trinius, P., Willems, C., and Holz, T. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur. 19*, 4 (Dec. 2011), 639–668.

[104] River, W. Applying multi-core and virtualization to industrial and safety-related applications. http://leadwise.mediadroit.com/files/8535WP_Multicore_for_Industrial_and_Safety_Feb2009.pdf, February 2009.

[105] Rössler, J., Zeller, A., Fraser, G., Zamfir, C., and Candea, G. Reconstructing core dumps. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (2013), pp. 114–123.

[106] Roussev, V. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI* (Berlin, Heidelberg, 2010), K.-P. Chow and S. Shenoi, Eds., Springer Berlin Heidelberg, pp. 207–226.

[107] Santamarta, R. *SATCOM Terminals: Hacking by Air, Sea, and Land.* IOActive, Inc., 2014.

[108] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[109] 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.

[110] Sebastián, M., Rivera, R., Kotzias, P., and Caballero, J. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2016), Springer, pp. 230–253.

[111] 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.

[112] Simmons, C., Ellis, C., Shiva, S., Dasgupta, D., and Wu, Q. Avoidit: A cyber attack taxonomy. University of Memphis, Technical Report CS-09-003, 2009.

[113] Soltani, M., Panichella, A., and van Deursen, A. A guided genetic algorithm for automated crash reproduction. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), pp. 209–220.

[114] Soltani, M., Panichella, A., and Van Deursen, A. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering* (2018), 1–1.

[115] Stakhanova, N., Couture, M., and Ghorbani, A. A. Exploring network-based malware classification. In *6th International Conference on Malicious and Unwanted Software* (2011), pp. 14–20.

[116] Suarez-Tangil, G., Conti, M., Tapiador, J. E., and Peris-Lopez, P. Detecting targeted smartphone malware with behavior-triggering stochastic models. In *Computer Security - ESORICS 2014* (Cham, 2014), M. Kutyłowski and J. Vaidya, Eds., Springer International Publishing, pp. 183–201.

[117] Tamás, C. Method for similarity searching in large malware repository. Master's thesis, Budapest University of Technology and Economics, 2018. Supervisor: Boldizsár Bencsáth.

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

[119] Upchurch, J., and Zhou, X. First byte: Force-based clustering of filtered block n-grams to detect code reuse in malicious software. In *8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)* (2013), pp. 68–76.

[120] Upchurch, J., and Zhou, X. Malware provenance: code reuse detection in malicious software at scale. In *11th International Conference on Malicious and Unwanted Software (MALWARE)* (2016), pp. 1–9.

[121] Vignau, B., Khoury, R., and Hallé, S. 10 years of iot malware: A feature-based taxonomy. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (July 2019), pp. 458–465.

[122] Vignau, B., Khoury, R., and Hallé, S. 10 years of iot malware: a feature-based taxonomy.

[123] Wang, P., Nguyen, H., Gu, X., and Lu, S. Rde: Replay debugging for diagnosing production site failures. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)* (2016), pp. 327–336.

[124] White, M., Linares-Vásquez, M., Johnson, P., Bernal-Cárdenas, C., and Poshyvanyk, D. Generating reproducible and replayable bug reports from android application crashes. In *2015 IEEE 23rd International Conference on Program Comprehension* (2015), pp. 48–59.

[125] Wilhelm, J., and Chiueh, T.-c. A forced sampled execution approach to kernel rootkit identification. In *Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2007), C. Kruegel, R. Lippmann, and A. Clark, Eds., Springer Berlin Heidelberg, pp. 219–235.

[126] Xu, Z., Zhang, J., Gu, G., and Lin, Z. Goldeneye: Efficiently and effectively unveiling malware's targeted environment. In *Research in Attacks, Intrusions and Defenses* (Cham, 2014), A. Stavrou, H. Bos, and G. Portokalidis, Eds., Springer International Publishing, pp. 22–45.

[127] Xuan, J., Xie, X., and Monperrus, M. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, Association for Computing Machinery, p. 910–913.

[128] Yampolskiy, M., Horvath, P., Koutsoukos, X. D., Xue, Y., and Sztipanovits, J. Taxonomy for description of cross-domain attacks on cps. In *Proceedings of the 2Nd ACM International Conference on High Confidence Networked Systems* (New York, NY, USA, 2013), HiCoNS '13, ACM, pp. 135–142.

[129] Yan, G., Brown, N., and Kong, D. Exploring discriminatory features for automated malware classification. In *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2013), DIMVA'13, Springer-Verlag, p. 41–61.

[130] 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).

[131] Ye, Y., Li, T., Chen, Y., and Jiang, Q. Automatic malware categorization using cluster ensemble. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2010), KDD '10, Association for Computing Machinery, p. 95–104.

[132] Yin, H., and Song, D. Analysis of trigger conditions and hidden behaviors. In *Automatic Malware Analysis: An Emulator Based Approach*. Springer New York, New York, NY, 2013, pp. 59–67.

[133] 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.

[134] Zhang, Y., Makarov, S., Ren, X., Lion, D., and Yuan, D. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 19–33.

[135] Zhu, B., Joseph, A., and Sastry, S. A taxonomy of cyber attacks on scada systems. In *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing* (2011), pp. 380–388.