Breaking Embedded Software Homogeneity with Protocol Mutations

Tongwei Ren[1], Ryan Williams[2], Sirshendu Ganguly[1], Lorenzo De Carli[1(✉)], and Long Lu[2]

[1] Worcester Polytechnic Institute, Worcester, MA 01609, USA
{tren,sganguly,ldecarli}@wpi.edu
[2] Northeastern University, Boston, MA 02115, USA
{williams.ry,l.lu}@northeastern.edu

Abstract. Network-connected embedded devices suffer from easy-to-exploit security issues. Due to code and platform reuse the same vulnerability oftentimes ends up affecting a large installed base. These circumstances enable destructive types of attacks, like ones in which compromised devices disrupt the power grid.

We tackle an enabling factors of these attacks: software homogeneity. We propose techniques to inject syntax mutations in application-level network protocols used in the embedded/IoT space. Our approach makes it easy to diversify a protocol into syntactically different dialects, at the granularity of individual deployments. This form of moving-target defense disrupts batch compromise of devices, preventing reusable network exploits. Our approach identifies candidate program data structures and functions via a set of heuristics, mutate them via static transformations, and selects correctness-preserving mutations using dynamic testing.

Evaluation on 4 popular protocols shows that we mitigate known exploitable vulnerabilities, while introducing no bugs.

Keywords: Software diversity · Protocol mutations · MTD

1 Introduction

Connectivity is now ubiquitous within smart and embedded devices—appliances such as light bulbs, power meters and industrial control systems; and in the near-future robot swarms, sensors and weapon systems. Such devices are already deployed in large numbers, with glaring security vulnerabilities [21,29,33,41,49], and outdated, hard-to-upgrade firmware [56]. Different devices may reuse the same components and platform [39], resulting in replication of vulnerabilities.

The situation presents analogies with the early 2000s, when worms like CodeRed spread uncontrollably, exploiting widely-installed, vulnerable internet-facing software [44,45]. The same factors are now resulting in botnets such as Mirai [16,31,43]. For now, large-scale embedded device compromise has resulted in attacks that—while large—remain within reach of traditional cyberdefenses.

F. Li et al. (Eds.): SecureComm 2022, LNICST 462, pp. 770–790, 2023.
https://doi.org/10.1007/978-3-031-25538-0_40

New attack models are however possible, and researchers are just beginning to understand them. Ronen et al. [50] demonstrated an IoT worm targeting smart public street lights. Possible attacks involve bricking devices, wireless jamming, and inducing epileptic seizures at large scale. Other works suggest that with a relatively small botnet of power-hungry IoT devices, an attacker could create a demand surge large enough to collapse a large power grid [57], or control energy pricing [55]. In the industrial domain, highly damaging attacks have already been observed [46]. And internet-connected robotic swarms, poised to becoming commonplace in robotics and military applications [18,33], will need safeguards too as the consequences of attacks are potentially catastrophic.

Even a single widespread vulnerability can enable attacks. For example, a single 0-day affecting Huawei home gateway devices enabled cybercriminals to create a 100,000-strong botnet in December 2017 [31]. It is therefore important to reason about defense-in-depth techniques that can protect devices against yet-unknown 0-day attacks. Authentication helps preventing basic attacks, but it is ineffective in case of default or easily guessable credentials [21], or credential stuffing. Anomaly-based intrusion detection systems (IDSs) suffer from low accuracy [27], while specification-based IDSs may miss stealth attacks [19].

To address this problem, we propose a novel form of moving-target defense for embedded/IoT, based on protocol mutations. We address one of the fundamental problems of large-scale IoT deployments: software homogeneity. It can arise when multiple deployments of devices all share the same vulnerable network-facing code. We support the diversification of application-level network protocol implementations, resulting in dialects which can be deployed at the granularity of individual installations. Dialects can be made mutually incompatible, thus communication with a device without knowledge of its mutations is impossible. This approach works by statically analyzing and mutating protocol source code.

Our work is not meant to replace security best-practices; rather, it complements them. It prevents one-size-fits-all exploits, thus avoiding rapid attack propagation among embedded devices. Furthermore, it is easy to incorporate as mutations are largely automated. The approach is also desirable in contexts where encryption is difficult to deploy due to resource constraints. We note that our work is not applicable to consumer/home IoT scenarios, where applying a different mutation to each individual device sold quickly grows impractical. Instead, it targets *Self-Contained Critical Deployments*: military or infrastructure deployments of homogeneous devices, all managed by the same organization.

Achieving our goal entails solving a number of challenges. First, modifying sender and receiver of a network communication entails identifying code and data structures used to create and parse messages. Unfortunately, to the best of our knowledge no existing technique can identify and map this information. Thus, we devise our own technique, which we name PaCo (**Pa**rser/**Co**nstructor extractor), implementing a problem-specific static analysis heuristic. A second component, named Aloja[1], then selects candidate constructor/parser pair, apply mutations, and test the mutated implementation for correctness. The output is a protocol implementation which incorporates a set of safe mutations.

[1] In Catalan mythology, an *Aloja* is a mythical creature able to shape-shift into a bird.

Our results show that this approach results in mutation that are effective in blocking paths to successful attack completion. **Overall contributions:**

1. We propose a static analysis technique to identify message-related structures and functions in network protocol implementations.
2. Based on the technique above, we propose static program transformations to inject symmetric mutations in message generators/parsers.
3. We thoroughly evaluate the techniques above on **4** protocols. We achieve **93%** accuracy in identifying relevant functions. Mutations successfully block **all** reproducible CVEs we identified, and do not affect correctness.
4. We release a software artifact implementing our proposed techniques, enabling analysis and further experimentation by the community (https://osf.io/9gc3n/?view_only=6da059eab07f4ffe934d6b59b49fee2b https://github.com/TongweiRen/Aloja).

2 Background

2.1 Target Scenarios

We target a style of IoT and embedded systems that we term **S**elf-contained **C**ritical **D**eployments (SCDs): deployments where (i) the set of devices belonging to the deployment is known a priori; (ii) the devices perform a task where loss of function can lead to significant disruption; and (iii) an attacker with incentives to cause disruption can communicate to the deployment. SCDs arise in military and infrastructure-related settings. Compared to the consumer domain, SCDs are self-contained within a single organization, so backward compatibility is not critical. Furthermore, they are expected to survive for an extended period while performing critical functions, which makes the (small) additional effort to introduce mutations acceptable. We review two example scenarios.

Autonomous Robotic Swarms. A recent DARPA BAA, OFFensive Swarm-Enabled Tactics (OFFSET), envisions a class of autonomous swarms of unmanned aircraft and/or ground systems used to accomplish missions in urban environments [20]. US Army also funded the MAST project, aimed at autonomous collections of intelligence-gathering robots [1]. A major issue with robotic swarms is attacks where malicious robots can be injected into the swarm [33]. Consequences of a cyberattack involve equipment damage, and failure to complete the task.

Infrastructure/Industrial. Increasingly, public and industrial spaces incorporate network of internet-connected embedded appliances, such as street lights and security cameras. These deployments are characterized by a large number of identical devices within the same network. Attacks may create widespread disruption: if a firmware vulnerability exists, then *all* devices are susceptible to it. In past work, Ronen et al. demonstrated a building-scale attack, where all the smart lights in a large building fall under control of an attacker [50].

2.2 Software Diversity in SCDs

Best-practice security measures such as encryption and authentication can help mitigate the risk of attackers infiltrating SCDs. However, past experiences demonstrate that buggy code, leftover development accounts, and other high-level issues can allow the attacker to bypass these lines of defense [38]. Here, we propose the use of *software diversity* as an additional line of defense that can provide additional security when paired with best-practices, in a defense-in-depth approach.

Software diversity [30,40] consists in deploying, with each installation of a given software, a copy which is functionally identical, but differs in the implementation from any other copy. Differentiation is typically envisioned using specialized compilation [22] or binary rewriting techniques [26], in some cases aided by VMs [61]. Differentiation aims at preventing an attacker to devise a general exploit by analyzing a single copy of the program. Introducing variations forces the attacker to tailor their exploits to each copy of the program.

Diversity-based defenses however cannot be ported directly to the SCD domain. Any approach with large overhead—e.g., interposing a VM—has to confront the resource-constrained nature of many devices. Furthermore, some attacks exploit vulnerabilities at levels of abstraction high enough to be impervious to binary-level diversification (e.g. default factory passwords). Third, vulnerabilities are nearly exclusively triggered via network; it is therefore reasonable to deploy diversity within components that deal with network communications.

A relevant question is whether diversification at the granularity of SCDs—and not individual devices—provides enough variety to prevent standardized exploits from spreading. Based on the large number of deployments for popular embedded devices, we expect SCD-level granularity to still provide sufficient diversity.

2.3 Goals and Threat Model

Our goal is to introduce software diversity by injecting mutations in the syntax of network protocol messages, thus preventing standardized exploits. Since attacks infect victims via malicious network input, we mutate protocols so that different deployments speak different *protocol dialects*. Mutations can be made incompatible: valid messages within one dialect are rejected by the network message parser in another. By focusing on early rejection of input, our approach is practical against both low-level binary exploits and high-level logic bugs.

We do not target protocols at layer-IV and below (e.g., WiFI, IP, 6LoWPAN, TCP), as those are typically implemented within the operating system and/or hardware. Instead, we focus on implementations of middleware protocols, such as Cyclone DDS [11]. These typically provide services like publisher/subscriber communication, and have wide applicability within the SCD domain. For example, the DDS protocol is used in domains as diverse as smart cities [10], robotics [42] and military applications [59].

In our current work, we mostly focus on *off-path* attackers. The attacker scans the internet for vulnerable devices (e.g., using vulnerability search engines [8]).

They commandeer any vulnerable device which replies to their probes. For example, the original Mirai botnet was built using similar techniques [16]. The attacker can send traffic towards potential victims and receive replies, but cannot observe the victim's communication with other nodes.

We also discuss *on-path attackers*. In addition to sending and receiving traffic to/from victim, this attacker can observe communications between non-compromised devices prior to the attack. The attacker may use its on-path capability to reverse-engineer wireless communications and identify vulnerabilities, inject messages, and batch-compromise devices.

In general, off-path attackers can be targeted using *static mutations*, and on-path attackers using *dynamic mutations*. We discuss both in the following.

2.4 Possible Mutation Types

We define different categories of protocol mutations, each being relevant to different types of attackers.

Static Mutations. A static mutation is statically embedded into the protocol firmware at compile time. A given mutated binary always exhibits the same mutation. The mutation behavior can evolve over medium time scales, by periodically recompiling the firmware (e.g., with firmware updates).

Static mutations target off-path attackers, who must guess the particular set of mutations in order to communicate. With an appropriately large set of mutations and their parameters, the effort required to break into the device is increased many-fold. Consider an attack which is carried over a sequence of N messages. Assume each message is independently mutated by a randomly selected mutation[2] from a set S s.t. $|S| = M$. Without any additional information, an off-path attacker must guess the correct mutation by trial-and-error; worst-case and expected number of tries are respectively MN and $\frac{M+1}{2}N$. A multiplicative increase in the attack complexity raises the bar against casual attackers, and has the advantage of simplicity. In this paper, we focus on static mutations.

Dynamic Mutations. A dynamic mutation is a one which can be reparametrized or disabled without recompiling the program. Dynamic mutations become necessary with an on-path attacker, who may be able to reverse-engineer mutations, by comparing unmutated protocol executions to the mutated ones in the target network. This attack can be mitigated by deploying a set of dynamic mutations, which evolve according to a *mutation schedule*. Dynamic mutations introduce additional complexity: two communicating peers must synchronize their mutation schedules so that they are always speaking the same protocol dialect, without leaking details of the schedule to the attacker. Our implementation of mutations exports an API through which individual mutations

[2] Note that a parametrized mutation with an n-bit parameter can be seen as a set of 2^n possible distinct mutations.

can be enabled/disabled at run-time, which provides a foundation for dynamic mutations. However, we leave a full implementation of this concept to future work.

2.5 One-Size-Fits-all Exploits

Oftentimes protocol implementations in the wild are affected by various types of bugs in the parsing logic. We performed an extensive review of historical parsing-related security bugs from six protocol implementations: Mosquitto MQTT [3], Wakaama [9], MQTT-C [6], Cyclone DDS [11], OpenDDS [7], and DSVPN[4]. Results show that these vulnerabilities can be categorized as follows: incorrect buffer sizing, lack of sanitization, and invalid/improper assertions. For example, a buffer sizing issue reported in CVE-2017-7651 causes a Mosquitto broker to crash after receiving a crafted packet. CVE-2017-7653 causes a Mosquitto broker to disconnect other clients, upon reception of an attack message with an invalid UTF-8 string. This bug is due to lack of string checking. Our approach focuses on early rejection of input, with the goal of containing this style of attacks.

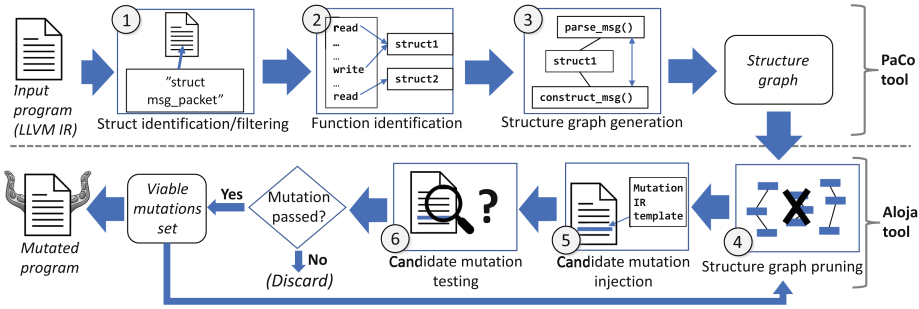

Fig. 1. Overview of Aloja

3 Approach Overview

Our approach receives as input the source of a network protocol implementation, such as Mosquitto MQTT [3]. We assume that a single codebase contains both components necessary to create and parse messages. We first analyze the code to identify program components (data structures and functions) which are involved in creating and parsing network messages. This task is performed by the PaCo tool, discussed in Sect. 3.1. The information returned by PaCo is then used to identify suitable locations for mutations, and to injects the actual mutations. This is performed by the Aloja tool, described in Sect. 3.2. The overall workflow is described in the following and outlined in Fig. 1.

3.1 PaCo: Identification of Relevant Program Components

First, the PaCo tool identifies program functions and structures whose purpose is to construct and parse network messages. PaCo's overall algorithm receives as input a program implementing a network protocol of interest, and return a structure graph describing *message-carrying structs* and *protocol constructors and parsers*, which we define below. Our implementation is based on the LLVM compiler toolkit, and works at the level of LLVM IR.

Definition 1 (Message-carrying struct). *A message-carrying struct is a composite data type which represents the structure and content of a network protocol message as a sequence of contiguous binary fields, to be serialized when the message is sent. Such a struct s consist of a sequence of field types* $s = f_1, ..., f_n$.

The notion of message as a sequence of binary fields is consistent with protocol reverse-engineering literature [28], and captures a large number of application-level protocols used in the embedded/IoT space. Based on this assumption, the definition of message constructor and parser follows:

Definition 2 (Constructor and Parser). *Consider the IR-level representation R of the implementation of a protocol P, and the set S of message-carrying structs in R. A message constructor function (constructor)* $f_C : S \rightarrow P$ *is a function in R which receives as input a struct argument and returns a valid message in the protocol P. Similarly, a message parser function (parser)* $f_P : P \rightarrow S$ *is a function which receives as input a message and returns a struct.*

Unambiguous identifying the entities above in R is challenging, due to the variety of parsing and construction techniques used in different network protocols. Rather than attempting to model the semantic of all possible protocol implementations, PaCo's algorithm is heuristic and returns an approximation of the correct sets of structs, constructors and parsers. The approximation is neither sound nor complete; however, Sect. 4 shows that it is highly accurate.

PaCo Workflow. First, PaCo collects all struct types appearing in the program, and filters the result to remove standard library structs (e.g., `__sigset_t`) unrelated to network messages. Further filtering retains only structs passed as input or output to functions that either (i) read or write data from the network; or (ii) perform memory copy (e.g., `memcpy`). Including the latter is necessary as oftentimes protocol code will not send and receive directly from structs, but copy parts of them into separate buffers. These operations, depicted in Step **1** of Fig. 1, result in a set S of candidate message-carrying structs.

PaCo then proceeds to collect functions operating on structs in S and building a *structure graph*. The algorithm iterates over all functions in the program, collecting each function that either reads or write to a variable whose type is in S (Step **2** in Fig. 1). This result in a set F of relevant functions. Intuitively, the set F includes candidate constructors and parsers in R.

Finally, PaCo builds a structure graph $G = (V, E)$ as follows. First, it defines the set of vertices V as $S \cup F$. Then, it establishes an edge between each function node in F and the structs in S that the function reads and/or writes (Step **3**).

Table 1. ALOJA's mutation library

Mutation	Description
Global/const refactoring	Remap values of enum fields/globals/consts
Mutate field value	Linearly combine field value with parameter
Encrypt field value	Encrypt field value with pre-determined cipher/key

3.2 ALOJA: Deployment of Mutations

ALOJA uses the structure graph G to inject *symmetric mutations* in constructor and corresponding parser functions.

Definition 3 (Mutation). *Consider a message-carrying struct $s \in S$, defined as in Definition 1. Further consider a constructor and a parser function $f_C, f_P \in F$ which operate on s. A mutation is a transformation of f_C and f_P in such a way that the mutated functions f'_C, f'_P exchange messages in a format $s' \neq s$ different from their non-mutated counterparts, but the result of the message exchange is functionally identical.*

Mutations must be *invertible*, i.e., it must be possible to reverse any transformation applied to generated messages when those are received. This excludes mutations that hash message fields, and similar applications of one-way functions. Any mutation which removes data from messages is also non-invertible.

Intuitively, ALOJA embeds a given mutation function into a constructor f_C, and the inverse mutation into the corresponding parser f_P. A desirable property of invertible functions, or bijections, is that they are *composable*; i.e., the composition of invertible functions is also invertible. It is thus possible to inject multiple mutations by composing the corresponding mutation functions.

In practice, we meet the requirement above by restricting ALOJA to mutations which apply invertible transformations to individual fields. Supported mutations, satisfying the property above, are detailed in Table 1. Our implementation, based on the injection of mutation templates into protocol code, makes it easy to expand this set with arbitrary additional mutations.

ALOJA Workflow. ALOJA identifies candidate locations for mutations by analyzing the structure graph G. First, it marks every function operating on any struct member in G as parser, constructor, or both based on whether the function reads/writes it. It also discards any struct member whose compiler-level type is not compatible with any mutation. Then, applies a heuristic to remove parser and constructor functions that are unlikely to directly affect wire-level protocol messages. First, ALOJA generates the function-level callgraph. Furthermore, it removes each constructor (parser) which has a direct edge to (from) another constructor (parser). The empirical insight is that values that affect network messages are typically written (read) by low-level (high-level) utility functions, which appear as leaves (non-leaf nodes) in the callgraph. This is represented in Fig. 1, Step **4**. The output of this step is a pruned structure graph G'. We remark

that filtering candidates is a performance optimization, but it is not necessary for correctness—thus, we believe an heuristic approach is appropriate. Our testing step, described below, can effectively remove mutants that involve incorrectly selected parsers/generators. We further discuss the performance benefits of filtering in Sect. 4.5.

In the next step, ALOJA build all possible mutation candidates, where a candidate $c = (f_C, f_P, M)$ is a tuple including a constructor f_C and a parser f_P in G' operating on the same struct s, and a mutation compatible with the type of s. To deploy a candidate mutation, ALOJA wraps the last write to the target struct member in s within the constructor f_C with a template which mutates s prior to writing. Similarly, the parser f_P is mutated by injecting a wrapper implementing the reverse mutation before the first field read (Step **5**).

For the same mutation strategy, our implementation will insert different IR code into the application based on the type of field we choose to mutate. For example, the code for changing `int32` field and `int64` field is syntactically different, but semantically same. This is to make sure our mutation will not change the field length and cause errors.

Next, the mutated program is tested to evaluate correctness. This step is the only one requiring (offline) user involvement, to specify a command executing a test procedure to verify correctness. Note that the procedure may simply run unit tests shipped with the codebase. For each mutation, ALOJA first runs a mutated server and client to check whether they can communicate correctly; then, it runs a mutated server and an unmutated client to ensure that the communication fails (Step **6**). ALOJA returns all mutations that pass both tests.

4 Evaluation

In this section, we evaluate ALOJA against the following experimental questions:

- **Question #1: can PACO correctly identify message-related program structures and functions?** In Sect. 4.2, we show that PACO can identify relevant program elements with high accuracy.
- **Question #2: do mutations lead to a measurable change in the difficulty to exploit a vulnerability?** In Sect. 4.3, we show that ALOJA's mutations can mitigate reusable attacks.
- **Question #3: can ALOJA correctly introduce mutations without affecting program correctness?** Extensive testing, discussed in Sect. 4.4, shows that ALOJA did not introduce any new bug.
- **Question #4: are ALOJA filtering heuristics successful in limiting the number of mutations to be tested?** Sect. 4.5 shows that filtering heuristic lead to up to a **41%** reduction in the number of tested mutations.
- **Question #5: do introduced mutations generate overhead in the execution of mutated protocols?** In Sect. 4.6, we show acceptable compile-time and run-time overhead, thus our mutations are practical.

Table 2. Characterization of codebases

Program	Protocol	# LOCs	# Functions
Mosquitto [3]	MQTT	79,859	338
Wakaama [9]	LwM2M	28,040	302
MQTT-C [6]	MQTT	12,845	98
Cyclone DDS [11]	DDS	109,954	2,146

Table 3. Accuracy of PaCo in identifying message-related structures

Program	True positives (Positives)	True negatives (Negatives)	Accuracy
Mosquitto	2 (3)	26 (26)	96.6%
MQTT-C	1 (1)	17 (17)	100%
Wakaama	5 (5)	18 (18)	100%
CycloneDDS	42 (51)	291 (300)	97.3%
Overall	50 (59)	352 (361)	97.8%

4.1 Implementation and Dataset

We created a prototype implementing the full pipeline described in Sect. 3. It is implemented as a set of LLVM passes and Python modules used to statically analyze IR files, perform mutation injection, and run the mutation test process. In total, our pipeline consists of 1183 lines of C/C++ code (LLVM passes) and 7961 lines of Python code (IR analysis and test automation).

Dataset. Table 2 describes the codebases used for the experiments. We selected these 4 projects as they are implementations of representative protocols commonly used in the IoT realm. Mosquitto [3] is an implementation of the popular MQTT protocol, while MQTT-C [6] is another implementation stripped of all extraneous features to provide a minimal install. We also evaluate Wakaama [9], a C implementation of the LwM2M machine-to-machine protocol, and Cyclone DDS [11], a C implementation of the DDS protocol. Although all applications we evaluated are middleware protocol applications, our system can be used for any upper-layer protocol applications using struct to construct messages.

4.2 Structure Graph Generation

Methodology. In this section, we evaluate the effectiveness of PaCo in identifying message-related functions and structs. In order to identify false positives and negatives, we performed a manual analysis of each codebase and compared the correct sets of functions and structures to those generated by PaCo. Manual analysis involved going through every function and struct, and verifying based on their name and functionality if they were, in fact, parser- or constructor-related. We envision PaCo as a general tool for code understanding; therefore

Table 4. Accuracy of PaCo in identifying message-related functions

Program	True positives (Positives)	True negatives (Negatives)	Accuracy
Mosquitto	10 (11)	203 (203)	99.5%
MQTT-C	6 (6)	57 (57)	100%
Wakaama	32 (32)	60 (66)	93.9%
CycloneDDS	26 (39)	89 (102)	81.6%
Overall	74 (88)	409 (428)	93.6%

we use a broader notion of "message-related" than that used by Aloja. We consider a function/struct message-related if it performs processing/stores data which is directly or indirectly used to construct and/or parse network messages.

Results-Structure Identification. Table 3 summarizes structure identification accuracy. True Positives/Negatives are the numbers of network-related/non-network-related functions PaCo successfully identified. Positives/Negatives represent the numbers of actual network-related/non-network-related functions in each codebase. Structure identification is heuristic and based on how a struct is used. False negatives come from network-related structs not being directly used for memory copy and network operations. False positives occur due to PaCo heuristics being misled by the presence of syscalls that are frequently, but not always, network-related, e.g. redundant `recv` functions when the application actually uses `read` to receive messages. Overall, results show high accuracy across the codebases (**97.8%**).

Results - Function Identification. Table 4 summarizes results regarding the accuracy of function identification across our four evaluation codebases. Overall, PaCo achieves **93.6%** accuracy. Analysis of mistakes indicates that they occur due to mislabeling of the corresponding structs, as discussed above.

Overall, we conclude that PaCo's heuristics are effective in identifying message-related structures and functions, providing actionable information to Aloja.

4.3 Exploit Mitigation

Methodology. For this experiment, we compare the behavior of unmutated compiled binaries with mutated ones. We identified reproducible vulnerabilities in the protocols of interest, by picking older releases with known vulnerabilities, and mutating them using Aloja. The mutation strategy we used for our experiments was mutating a field value as shown in Table 1, row 3. This process resulted in two versions of a vulnerable protocol implementation, without and with the mutation. We first verified that the vulnerability could be successfully reproduced in the non-mutated client/server. Then, we used the same client and sent the same packets, as we did before, to the mutated server, to check that the vulnerability could no longer be triggered. Because our mutations targets

parser/constructor functions, we look specifically at parser-/constructor-related vulnerabilities that are triggerable by malformed network input. For example, CVE-2017-7653 causes a Mosquitto broker to disconnect other clients, upon reception of an attack packet containing an invalid UTF-8 string.

Collected Vulnerabilities. For Mosquitto, we did an extensive review of relevant Common Vulnerabilities and Exposures (CVE) reports. We found 17 vulnerability reports of which 7 are parser-/constructor-related. We only show the results of three CVE reports which we could reliably reproduce, including CVE-2019-11779, CVE-2018-12543 and CVE-2017-7653. We could not reliably reproduce the remaining ones (e.g. CVE-2019-11778), as they are caused by heisenbugs. For Wakaama, we reliably reproduced one relevant CVE report, CVE-2019-9004. For MQTT-C and Cyclone DDS, we did not find recent relevant CVE reports.

Results. In the case of CVE-2019-11779, the reception of the attack packet in unmutated Mosquitto triggers the vulnerable function `mosquitto_sub_topic_check`. The same attack against the mutated version results in the packet being dropped. The vulnerability is not triggered, and the unmutated attacker's client gets forcibly disconnected. Our experiments for CVE-2018-12543, CVE-2017-7653, and CVE-2019-9004, also resulted in successful exploit in the non-mutated version, and failed exploit in the mutated one. These results show that ALOJA's mutation approach is effective in mitigating network-based exploits.

4.4 Correctness

Methodology. In this section we evaluate the correctness of mutations applied by ALOJA. It is important to ensure that the mutations do not break any underlying functionality. In order to do so, we run unit tests against the unmutated and mutated version of each protocol, and compare the results of the two runs. The mutation strategy we used for our experiments is mutating an existing field value, shown in Table 1, row 2. All protocols include high-quality developer-provided unit tests which we use for this purpose. We emphasize that these are **not** the tests we used for selecting useful mutants. To avoid biasing the correctness evaluation, for the latter purpose we develop our own test scenarios.

We also performed an in-depth case study on Mosquitto, by designing and building a fuzzing MQTT client based on the Boofuzz fuzzer [5]. The fuzzer generates protocol messages and directs them to a mutated Mosquitto binary. To ensure extensive coverage, we do not generate completely random input, but messages with a structure approximating MQTT's specifications and sessions.

Results. Table 5 shows that 3 out of 1233 test cases failed. We manually analyzed each failed test; in all cases, test failures were due to the tests expecting protocol messages in the original (i.e., non-mutated) format. We ruled out these failures as benign, as our mutation strategy is specifically designed to trigger failures like these. Furthermore, our Mosquitto fuzz tester ran for 64.22 min without triggering any unexpected behavior. Overall, these results suggest that ALOJA does not introduce any new bugs or vulnerabilities into the codebase it mutates.

Table 5. Results of correctness evaluation tests.

Program	# Tests	# Failed tests (benign)
Mosquitto	207	2 (2)
Wakaama	71	0 (0)
MQTT-C	18	1 (1)
Cyclone DDS	937	0 (0)

Table 6. Impact of heuristic on mutation generation

Program	W/O Heuristic (Useful))	W/Heuristic (Useful)
Mosquitto	242(2)	142(2)
MQTT-C	116(7)	73(5)
Wakaama	340(3)	235(3)
Cyclone DDS	969(6)	795(2)

4.5 Impact of Mutation Filtering Heuristics

Methodology. Aloja uses callgraph analysis to filter constructor and parser functions unlikely to directly operate on message content (discussed in Sect. 3.2). It is important to determine whether such heuristics (i) lead to a reduction in the number of possible mutations to be tested; and (ii) do not lead to loss of potentially useful mutations. In order to evaluate the impact of heuristics, we ran Aloja twice on each codebase, first with the heuristic deployment, and then deactivated. We compared the runs on two aspects: number of useful mutants and number of overall generated mutants.

Results. Table 6 show the number of useful and overall mutations across the three codebases. The heuristic causes a best-case reduction of **41.3%** (Mosquitto) in the number of mutants, and an overall reduction of **25.3%**. Only **6 out of 422** removed mutations are useful, which amount to losing **33.3%** of useful mutations. We conclude that the filtering heuristic selects and remove incorrect mutations with reasonable accuracy.

4.6 Performance Impact of Mutations

Methodology. To evaluate the overhead incurred due to running a mutated protocol, we examined two aspects. The first is the compile-time overhead due to running PaCo and Aloja, measured by timing the overall compilation without and with the mutation process. We do not report detailed results on code size increase due to mutation, as such overhead remains small—between **0.1%** and **1%**—for all codebases.

Table 7. Incurred compile time overhead. PaCo overhead includes structure graph construction. Aloja's overhead is dominated by testing of mutants.

Program	w/o mutations	w/ mutations	PaCo	Aloja
Mosquitto	3 m 21 s	11 m 36 s	10 s	8 m 05 s
Wakaama	2.33 s	15 m 21.33 s	24 s	14 m 55 s
MQTT-C	1.85 s	8 m 49.41 s	0.56 s	8 m 47 s
Cyclone DDS	1 m57 s	152 m12 s	13 m 18 s	138 m 54 s

The second is the run-time overhead due the additional computation performed by injected mutation templates. In order to estimate this, we measured the latency overhead when sending batches of 10 messages per client, with a fixed size of 10KB each, via the respective protocol with a set number of clients in the network. Note that the impact of the number of clients in the network on protocol performance depends on the design of the protocol itself. Therefore, introduced overhead cannot be compared across different protocols; however, the results still highlight general trends in the overhead introduced by mutations. We evaluated this overhead using 10, 100, and 500 clients for each protocol. As deploying a device network of this size on an experimental testbed is impractical, we simulated the setup by running the server and the client instances on a dedicated machine. We acknowledge this approach is only an approximation of an actual deployment; however we believe it is sufficient to derive high-level conclusions about mutation overhead. All performance evaluations were run on an Ubuntu 20.04 VM with 4 3.7 GHz cores and 10 GB of RAM.

Results-Compile Time. Across our 4 codebases, it took, on average, an additional **45.7** min to run the mutation-enabled compilation process. This includes all stages of PaCo and Aloja. Table 7 compares the compile times achieved by clang to that of our process, which also uses clang but applies mutations prior to binary generation. Most of increased overhead is due to testing mutations in Aloja. For each possible mutation, Aloja must run the mutated application twice. The significant compilation time for Cyclone DDS indeed results from the fact that its large codebases induces many candidate mutants, than must then be vetted (on average, testing a mutant requires **10 s**). The process could be sped up by better mutation filtering heuristics, which we leave as future work. Also, multiple mutants could be tested in parallel, which would result in significant improvement on modern multicore platforms. Finally, we emphasize that the mutation overhead only need to be incurred once at compile time.

Results-Execution Time. Table 8 shows the overhead in message send latency introduced by Aloja. The overhead ranges from negligible to significant, although the relative overhead generally increases sublinearly with the number of clients, and notably decreases in the case of Wakaama when going from 100 to 500 clients! This fact in particular lead us to suspect that 100-client Wakaama

Table 8. Incurred overhead of running **n** = 10, 100, and 500 clients sending 10 KB messages in the original vs. mutated system

Program	10 clients	100 clients	500 clients
Mosquitto (**Mutated**)	3 ms (**3 ms**)	47 ms (**58 ms**)	160 ms (**200 ms**)
Wakaama (**Mutated**)	80 ms (**122 ms**)	120 ms (**230 ms**)	270 ms (**360 ms**)
MQTT-C (**Mutated**)	15 ms (**15 ms**)	32 ms (**40 ms**)	120 ms (**190 ms**)
Cyclone DDS (**Mutated**)	14 ms (**15 ms**)	139 ms (**149 ms**)	699 ms (**851 ms**)

results may be affected by an experimental artifact, but were not able to trace the source of the deviation. Future work should focus on further optimizing mutations and better understanding their performance impact.

5 Discussion

Mutation vs Encryption. A possible way to diversify a protocol is to retrofit it with encryption, and vary encryption keys across deployments. We deliberately decided to design a more general approach based on field-level mutations, for the following reasons. Field-level mutations can be retrofitted automatically during compilation, and do not change protocol state machine, packet size and structure, thus maximizing compatibility with existing network infrastructure and middleboxes. It is also worth noting that, if desired, field-level mutations can be used to deploy encryption (supported by ALOJA). However, they are not limited to it, and can also implement more lightweight forms of obfuscation. We believe this to be useful to control the performance impact of mutations. For smart sensors, which have to operate for extended periods without battery replacement, even a small increase in computation may translate in significant reduction in device life. Summarizing, our goal is to provide software diversity, not data secrecy. Consistently with the end-to-end principle [52], we believe this is best served by evaluating a range of syntactic mutations, and let application designers choose the most appropriate.

Mutation-Agnostic Attacks. Other forms of mutation-based moving target defenses have been shown to be vulnerable to mutation-agnostic attacks [51]. In our context, such an attack would consist of a message which triggers a vulnerability regardless of the mutations. This may happen for example if the victim parses each message field iteratively, and relevant mutations only affect fields which are parsed after the malicious data. These attacks can be mitigated by carefully choosing mutations, which always minimize the amount of code executed in response to an non-mutated message.

Integration into Embedded Development Workflow. A relevant question is whether incentives exist for embedded developers to integrate mutations into firmware, since manufacturers have limited incentive to improving security. As

discussed in Sect. 2.1, we target self-contained deployments managed by same organization. Firmware for mission-critical devices such as robot swarms or military is custom-developed and specified by contract, which simplifies requesting mutation technology. Cheaper white-label devices, such as camera, may ship with closed-source firmware. There, applying mutations may require re-flashing devices with Open-Source, customizable firmware (e.g., OpenIPC [12] for cameras).

Limitations of Dynamic Analysis. The dynamic testing techniques we employ, such as unit tests, cannot guarantee correctness. However, we decided to use them due to the intrinsic limitations of the alternative, which is static analysis. Network protocol implementation exhibit a great variety of programming patterns, which makes statically inferring mutation locations challenging. This is exacerbated by the fact that core static analysis algorithms are in general undecidable, and have precision limitations [54]. Thus, we consider dynamic analysis as an acceptable trade-off.

6 Related Work

Moving Target Defense (MTD). Traditional MTD strategies can be divided into two categories: operating system (OS)-, network-level. As MTD is a large area of research, we only discuss selected examples. At the OS level, Seibert et al. [53] and Hu et al. [34] use Address Space Layout Randomization (ASLR) and Instruction Set Randomization (ISR) to prevent memory corruption vulnerabilities and low-level code injection attacks. Pappas et al. [47] and Wartell et al. [60] introduce schemes to prevent memory disclosure issues by randomizing the data and code segments of each application [64]. At the network level, Al-Shaer [13] proposes an architecture to enable network configuration changes without disrupting network operations. Haadi Jafarian et al. [37] present a software-defined network (SDN)-based MTD strategy that mutates IP addresses, while maintaining configuration integrity. Huang and Ghosh [35] present an MTD scheme to protect web services, by creating a set of diverse offline virtual servers to replace online servers according to the rotation schedule. OS-level mutations do not prevent many kinds of relevant high-level attacks (such as default password reuse). Traditional network-level mutations affect the network configuration and are orthogonal to ours; they can be seen as an additional possible layer of defense. We separately discuss software-level mutations below.

Software Diversification. Software diversification is a popular MTD strategy to prevent application level attacks. Jackson et al. [36] propose a compiler-based technique which uses instruction set and register randomization to generate a large number of variants. This technique is designed for mobile apps. Franz [30] similarly discuss a mobile-oriented approach to generate a unique version of an app for each client which downloads it. Cabutto et al. [17] propose to store chunks of executable binary on a trusted server, dynamically downloading them

at execution time. However, this method cannot prevent existed vulnerabilities from being triggered since the attacker can still touch the vulnerable code. Wu et al. [62] present LLVM-based binary software randomization, which apply a number of IR-level transformations (e.g., instruction replacement) prior to compilation. This technique has limitations similar to OS-level MTD. Beurdouche et al. [15] propose a verified implementation of a TLS state machine that can be embedded into OpenSSL to change the overall state machine. Our system focuses on a broader set of protocol implementations. Collberg et al. [24] use a trusted server to generate diverse code variants, which are then dynamically installed within running clients. Our system injects the full mutation logic within the client, thus simplifying deployment. Cui and Stolfo [25] propose a host-based defense mechanism called Symbiotic Embedded Machines (SEM). They inject SEMs into host software as an additional component providing monitoring and defense. Compared with SEM, our method does not introduce an executable middleware and does not impose significant overhead on the target application. Pappas et al. [48] propose in-place code randomization, which breaks the semantics of gadgets used in return-oriented programming attacks. Our technique addresses a broader range of attacks.

Mutation Testing. Mutation testing, which aims at introducing errors in source code to ensure effectiveness of test cases, can also generate different mutants during the test process. Hariri et al. [32] present a toolset, SRCIROR, for achieving mutation testing at the C/C++ source code and LLVM intermediate representation (IR). Sousa and Sen [58] also present a LLVM IR-based mutation testing, which changes integer constants, to improve the generation of Transaction Level Modeling (TLM) testbenches. Although mutation testing applies mutations similar to those used in MTD, its goal is orthogonal. Evaluating the applications of ALOJA to mutation testing is an interesting direction for future work.

Parser/Constructor Function Identification. Polytracker [2] is an LLVM-based instrumentation tool for dynamic taint analysis, which we initially attempted to use within our project. However, we found that, although taint tracking can be used to locate parsing code, it is less suited for identifying message constructors. Besides, it requires users to provide sufficient and comprehensive inputs. Cojocar et al. [23] propose PIE, a methodology to identify protocol implementation parsers in embedded systems. Similar to Polytracker, PIE is also a parser identification approach and does not identify constructors. Bao et al. [14] and Yin et al. [63] present binary-analysis-based function identification methods. Compared to them, our function mapping leverages source code, and can provide a smaller but more precise function set which includes parser/constructor functions.

7 Conclusion

In this paper, we described a technique for injecting mutations into implementations of embedded/IoT-oriented network protocols as a form of moving-target

defense. Our evaluation shows that we correctly identify message-generating and parsing code and inject mutations which preserve functional correctness of a protocol. Furthermore, mutations are effective in preventing one-size-fits-all exploits, and only introduce limited overhead. By automating program analysis and transformations necessary for mutations, our work provides an important foundation for moving-target defense based on protocol dialects.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. This project was supported by the Office of Naval Research (Grants#: N00014-18-1-2660; N00014-21-1-2492). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

1. Micro Autonomous System Technologies (MAST). http://www.mast-cta.org/
2. trailofbits/polytracker: An LLVM-based instrumentation tool for universal taint analysis. https://github.com/trailofbits/polytracker
3. Eclipse Mosquitto (January 2020). https://mosquitto.org/
4. DSVPN (February 2021). https://github.com/jedisct1/dsvpn
5. GitHub - jtpereyda/boofuzz (February 2021). https://github.com/jtpereyda/boofuzz
6. MQTT-C (February 2021). https://github.com/LiamBindle/MQTT-C
7. OpenDDS (August 2021). https://opendds.org/
8. Shodan (January 2021). https://www.shodan.io/
9. wakaama (February 2021). https://www.eclipse.org/wakaama/
10. Who's Using DDS? (January 2021). https://www.dds-foundation.org/who-is-using-dds-2/
11. CycloneDDS (2022). https://github.com/eclipse-cyclonedds/cyclonedds
12. OpenIPC (December 2022). https://openipc.org/
13. Al-Shaer, E.: Toward network configuration randomization for moving target defense. In: Moving Target Defense (2011)
14. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: BYTEWEIGHT: Learning to recognize functions in binary code. In: USENIX Security Symposium (2014)
15. Beurdouche, B., et al.: A messy state of the union: Taming the composite state machines of tls. In: IEEE S&P (2015)
16. Brian Krebs: Who Makes the IoT Things Under Attack? — Krebs on Security (October 2016). https://krebsonsecurity.com/2016/10/who-makes-the-iot-things-under-attack/
17. Cabutto, A., Falcarin, P., Abrath, B., Coppens, B., De Sutter, B.: Software protection with code mobility. In: ACM MTD Workshop (2015)
18. Cameron, L.: IoT Meets the Military | IEEE Computer Society (March 2017). https://www.computer.org/publications/tech-news/research/internet-of-military-battlefield-things-iomt-iobt
19. Caselli, M., Zambon, E., Sommer, R., Kargl, F., Amann, J.: Specification mining for intrusion detection in networked control systems. In: USENIX Security Symposium (2017)

20. Chung, T.: OFFensive Swarm-Enabled Tactics. https://www.darpa.mil/program/offensive-swarm-enabled-tactics
21. Cimpanu, C.: Hacker leaks passwords for more than 500,000 servers, routers, and IoT devices (January 2020). https://www.zdnet.com/article/hacker-leaks-passwords-for-more-than-500000-servers-routers-and-iot-devices/
22. Cohen, F.B.: Operating system protection through program evolution. Comput. Sec. **12**(6), 565–584 (1993)
23. Cojocar, L., Zaddach, J., Verdult, R., Bos, H., Francillon, A., Balzarotti, D.: Pie: Parser identification in embedded systems. In: ACSAC (2015)
24. Collberg, C., Martin, S., Myers, J., Nagra, J.: Distributed application tamper detection via continuous software updates. In: ACSAC (2012)
25. Cui, A., Stolfo, S.: Symbiotes and defensive mutualism: Moving target defense. In: Moving Target Defense, pp. 99–108 (August 2011)
26. Davi, L.V., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In: ASIA CCS (2013)
27. De Carli, L., Mignano, A.: Network security for home iot devices must involve the user: a position paper. In: FPS (2020)
28. De Carli, L., Torres, R., Modelo-Howard, G., Tongaonkar, A., Jha, S.: Botnet protocol inference in the presence of encrypted traffic. In: INFOCOM (2017)
29. Eduard Kovacs: Serious Vulnerabilities Found in Schneider Electric Power Meters | SecurityWeek.Com (March 2021). https://www.securityweek.com/serious-vulnerabilities-found-schneider-electric-power-meters
30. Franz, M.: E unibus pluram: Massive-scale software diversity as a defense mechanism. In: NSPW (2010)
31. Goodin, D.: 100,000-strong botnet built on router 0-day could strike at any time (December 2017). https://arstechnica.com/information-technology/2017/12/100000-strong-botnet-built-on-router-0-day-could-strike-at-any-time/
32. Hariri, F., Shi, A.: Srciror: A toolset for mutation testing of c source code and llvm intermediate representation. In: ACM/IEEE ASE (2018)
33. Higgins, F., Tomlinson, A., Martin, K.M.: Threats to the Swarm: Security Considerations for Swarm Robotics. Int. J. Adv. Sec. **2**(2&3) (2009)
34. Hu, W., et al.: Secure and practical defense against code-injection attacks using software dynamic translation. In: VEE (2006)
35. Huang, Y., Ghosh, A.: Introducing diversity and uncertainty to create moving attack surfaces for web services. In: Moving Target Defense, pp. 131–151 (August 2011)
36. Jackson, T., et al.: Compiler-generated software diversity. In: Moving Target Defense, pp. 77–98 (August 2011)
37. Jafarian, J.H., Al-Shaer, E., Duan, Q.: Openflow random host mutation: Transparent moving target defense using software defined networking. In: HotSDN (2012)
38. Kat Hall: Hyperoptic's ZTE-made 1gbps routers had hyper-hardcoded hyper-root hyper-password (April 2018). https://www.theregister.co.uk/2018/04/26/hyperoptics_zte_routers/
39. Krebs, B.: Naming & Shaming Web Polluters: Xiongmai - Krebs on Security (October 2018). https://krebsonsecurity.com/2018/10/naming-shaming-web-polluters-xiongmai/
40. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: Automated Software Diversity. In: IEEE S&P (2014)
41. Lewellen, T.: CERT/CC Vulnerability Note VU#800094 (September 2013). https://www.kb.cert.org

42. Maruyama, Y., Kato, S., Azumi, T.: Exploring the performance of ros2. In: EMSOFT (2016)
43. Merces, F., Remillano II, A., Molina, J.: Mirai Botnet Attack IoT Devices via CVE-2020-5902 (July 2020). https://www.trendmicro.com/en_us/research/20/g/mirai-botnet-attack-iot-devices-via-cve-2020-5902.html
44. Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: Inside the Slammer worm. IEEE Sec. Privacy **1**(4), 33–39 (2003)
45. Moore, D., Shannon, C., claffy, k.: Code-Red: A case study on the spread and victims of an internet worm. In: ACM IMW (2002)
46. Muncaster, P.: A Third of Industrial Control Systems Attacked in H1 2021 (September 2021). https://www.infosecurity-magazine.com/news/third-industrial-control-systems/
47. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: IEEE S&P (2012)
48. Pappas, V., Polychronakis, M., Keromytis, A.: Practical software diversification using in-place code randomization. In: Moving Target Defense (2013)
49. Pascu, L.: Multiple critical security flaws found in nearly 400 IP cameras - Bitdefender BOX Blog (June 2018), https://www.bitdefender.com/box/blog/ip-cameras-vulnerabilities/multiple-critical-security-flaws-found-nearly-400-ip-cameras/
50. Ronen, E., Shamir, A., Weingarten, A., O'Flynn, C.: IoT goes nuclear: creating a zigbee chain reaction. In: IEEE S&P (2017)
51. Rudd, R., et al.: Address oblivious code reuse: on the effectiveness of leakage-resilient diversity. In: NDSS (2017)
52. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Trans. Comput. Syst. (TOCS) **2**(4), 277–288 (1984)
53. Seibert, J., Okhravi, H., Söderström, E.: Information leaks without memory disclosures: Remote side channel attacks on diversified code. In: ACM CCS (2014)
54. Shapiro, M., Horwitz, S.: The effects of the precision of pointer analysis. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 16–34. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0032731
55. Shekari, T., Irvene, C., Beyah, R.: IoT Skimmer: Energy Market Manipulation through High-Wattage IoT Botnets - Black Hat USA 2020 (August 2020), https://www.blackhat.com/us-20/briefings/schedule/index.html#iot-skimmer-energy-market-manipulation-through-high-wattage-iot-botnets-20280
56. Simpson, A.K., Roesner, F., Kohno, T.: Securing vulnerable home IoT devices with an in-hub security manager. In: PerCom Workshop (2017)
57. Soltan, S., Mittal, P., Poor, H.V.: BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In: USENIX Security (2018)
58. Sousa, M., Sen, A.: Generation of tlm testbenches using mutation testing. In: CODES+ISSS 2012 (2012)
59. Wang, N., Schmidt, D.C., van't Hag, H., Corsaro, A.: Toward an adaptive data distribution service for dynamic large-scale network-centric operation and warfare (ncow) systems. In: MILCOM (2008)
60. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: ACM CCS (2012)
61. Williams, D., Hu, W., Davidson, J.W., Hiser, J.D., Knight, J.C., Nguyen-Tuong, A.: Security through diversity: leveraging virtual machine technology. IEEE Sec. Privacy **7**(1), 26–33 (2009)

62. Wu, B., Ma, Y., Fan, L., Qian, F.: Binary software randomization method based on llvm. In: 2018 IEEE International Conference of Safety Produce Informatization (IICSPI), pp. 808–811 (2018)
63. Yin, X., Liu, S., Liu, L., Xiao, D.: Function recognition in stripped binary of embedded devices. IEEE Access **6**, 75682–75694 (2018)
64. Zheng, J., Siami Namin, A.: A survey on the moving target defense strategies: An architectural perspective. J. Comput. Sci. Technol. **34**, 207–233 (2019)