Collapse Like A House of Cards: Hacking Building Automation System Through Fuzzing

Yue Zhang
Drexel University
Philadelphia, PA, USA
yz899@drexel.edu

Zhen Ling*
Southeast University
Nanjing, Jiangsu, China
zhenling@seu.edu.cn

Michael Cash
University of Central Florida
Orlando, FL, USA
mcash001@knights.ucf.edu

Qiguang Zhang
Southeast University
Nanjing, Jiangsu, China
qgzhang@seu.edu.cn

Christopher Morales-Gonzalez
UMass Lowell
Lowell, MA, USA
Christopher_MoralesGonzalez@student.uml.edu

Qun Zhou Sun
University of Central Florida
Orlando, FL, USA
QZ.Sun@ucf.edu

Xinwen Fu
UMass Lowell
Lowell, MA, USA
xinwen_fu@uml.edu

Abstract

Building Automation Systems (BAS) play a pivotal role in modern smart buildings, integrating sensors, controllers, and software to manage crucial functions such as HVAC, lighting, and more. The global smart building market is on the rise, underscoring the importance of securing BAS networks. This paper introduces the Building Automation System Evaluator (BASE), a specialized fuzzer designed to assess the security of BAS networks. BAS networks typically involve a BAS client communicating with a BAS server through BAS protocols (e.g., BACnet, KNX), each presenting unique challenges in BAS network fuzzing. These challenges encompass complex packet structures and sequencing in BAS protocols, closed-source clients with indeterminable code coverage, and unobservable server status with limited throughput. BASE automatically identifies protocol structures, dynamically instruments clients for code coverage analysis, and monitors responses for new coverage areas. Collected timestamps are used to estimate the input scan intervals of servers, optimizing throughput. We evaluated BASE on various BAS servers and clients, uncovering 13 new vulnerabilities. Furthermore, we present three attack case studies, highlighting the real-world security implications of these vulnerabilities in BAS systems, such as delayed fire detection, loss of climate control, and security breaches. We reported our findings to the respective vendors, who acknowledged the implications, and some have subsequently patched their systems based on our reports.

*Corresponding Author

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690216

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

IoT Security; Fuzzing; CPS Security; Building Automation System Security; Vulnerability Discovery

ACM Reference Format:

Yue Zhang, Zhen Ling, Michael Cash, Qiguang Zhang, Christopher Morales-Gonzalez, Qun Zhou Sun, and Xinwen Fu. 2024. Collapse Like A House of Cards: Hacking Building Automation System Through Fuzzing. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690216

1 Introduction

A building automation system (BAS) is a type of industrial control system (ICS) that consists of a network of interconnected sensors, controllers, and software used to implement various functions in a building, such as heating, ventilation, and air conditioning (HVAC), lighting, shading, and many other operations. BAS can be deployed in commercial, industrial, and residential environments. The integration of BAS within a building is commonly referred to as a "smart building." The global market for smart buildings is growing rapidly, and according to ReportLinker, by 2027, the BAS market is expected to reach $277 billion [36]

A BAS network typically comprises clients, servers, and various BAS devices. A BAS client, which serves as the user front-end controller (e.g., a laptop), usually has the capability to establish communication through BAS protocols (e.g., KNX [2], BACnet [11]) with a BAS server, constituting the backend of the BAS system responsible for connecting various BAS devices (e.g., lights and secure cameras). Communication between servers and clients, as well as between devices and servers, can be facilitated using wireless protocols such as WiFi. This essentially means that anyone

within the confines of a firewall possesses the capacity to both modify and monitor data across the BAS network. This heightened accessibility expands the potential attack surface of the BAS system, creating opportunities for vulnerabilities to manifest in the BAS client, the BAS server, or even the network protocol itself. Attackers positioned within the network can exploit these vulnerabilities to launch various attacks (e.g., Denial-of-Service [33], impersonation attacks [6, 29]).

To investigate the vulnerability landscape of BAS networks, employing fuzzing emerges as the natural choice—a widely adopted technique for autonomously identifying weaknesses and bugs (e.g., memory corruption) within applications. This can be achieved by generating IP packets and sending them to the appropriate BAS target (e.g., the BAS server or the client). However, fuzzing a BAS network presents unique challenges: (*i*) BAS IP packets exhibit complexity, comprising elements such as magic bytes, length fields, counter fields, and various other structures. Furthermore, some protocols have critical packet sequences that must remain unchanged. (*ii*) Fuzzing closed-source BAS clients makes it difficult to assess code coverage and determine when to stop. For example, many fuzzers assume programs will terminate after processing input, which cannot be directly used to calculate code coverage. (*iii*) Determining the operational state of the server and inferring its code coverage presents another formidable hurdle. This becomes even more complex as BAS servers may intermittently scan for inputs, thereby potentially constraining the efficiency and speed of the fuzzing process.

This paper presents **B**uilding **A**utomation **S**ystem **E**valuator (BASE), a fuzzer targeted specifically at BAS networks, which addresses these three challenges. First, BASE automatically identifies important structures within the protocol and the actual order of the packets, by *probing* the packets and comparing the new network responses from the BAS server to the expected responses. Second, BASE dynamically instruments the BAS client and calculates the code coverage of probed responses. The coverage score of the probed response is compared to the coverage score of the expected response to determine when to terminate. Finally, BASE functions as a BAS client, transmitting fuzzed requests to the BAS server while carefully monitoring responses that indicate the discovery of new coverage areas. Timestamp metadata from the corpus is used to estimate the input scan interval of devices (i.e., throughput), both during the probing stage and during the fuzzing stage.

We evaluate BASE on 11 BAS servers (i.e., 4 BACnet devices, 7 KNX devices), and 6 BAS clients. We discovered 13 new vulnerabilities. We conducted experiments on code coverage and comparisons with other fuzzers. Specifically, we evaluated our fuzzer's performance over 24 hours, during which it discovered 17,616 unique edges in the control flow graph and maintained an average execution speed of 138.91 executions per second. In terms of effectiveness, we compared our BASE with BooFuzz [5]. Particularly, Boofuzz sent 2,856,753 mutated messages with zero knowledge and found no vulnerabilities, highlighting the inefficiency of random fuzzing without targeted strategies. Even when Boofuzz was provided with knowledge of the packet structure, it still exhibited low performance with extensive message mutations. Our fuzzer surpasses others because it not only recognizes the structure of messages but also resolves the relationships between fields, enhancing its effectiveness in discovering vulnerabilities.

To demonstrate the implications, we also conducted three distinct attack case studies and successfully caused the devices under testing to crash. The first attack targeted a BASrouter and a connected sensor, rendering them unresponsive with a crafted BACnet/IP packet. The second disrupted a KNX damper by sending corrupted KNXnet/IP packets. The third attack on the ETS client used for KNX-based automation triggered memory consumption, leading to software unresponsiveness. These vulnerabilities and attacks can lead to severe consequences in building automation systems. For example, disruption of temperature or humidity sensors means the BAS can no longer accurately monitor and control the building's climate, causing occupant discomfort, potential damage to sensitive equipment, and even safety risks in extreme cases (e.g., overheating or freezing). Additionally, the BASrouter might serve as a gateway to other critical systems, such as security cameras and access control. Crushing those devices may allow an attacker to gain further access to the building's security infrastructure, compromising the safety of people and assets. Moreover, disrupting the fire alarm system could delay or prevent timely detection of a fire emergency, increasing risks to occupant safety and property damage.

Our major contributions are summarized as follows:

- **Novel Tool with Domain Insights.** We propose BASE, an automatic fuzzing tool to identify the bugs in BAS. We are the first to systematically assess security threats in BAS networks. BASE dynamically learns packet structures and sequencing, calculates code coverage for clients, and monitors server status. The collected timestamps are utilized to estimate the input scan intervals of devices, optimizing throughput.
- **New Bugs Impacting BAS Networks.** We evaluated BASE on 11 BAS servers, comprising 4 BACnet devices and 7 KNX devices, along with 6 BAS clients. We identified 13 new BAS vulnerabilities, including 8 client-side and 5 server-side bugs.
- **Practical Case Studies with Serious Impacts.** We conducted three attack case studies: crashing a BASrouter controller with a crafted BACnet/IP packet, disrupting a KNX damper controller with corrupted KNXnet/IP packets, and triggering memory consumption in the ETS client, causing software unresponsiveness in KNX-based building automation. These vulnerabilities may pose serious threats such as delayed fire detection, loss of climate control, and security breaches.

Responsible Disclosure: We upheld the highest ethical standards during the launch of our fuzzing testing and attacks. First, we conducted all experiments and attacks exclusively within a controlled environment on our own devices to prevent any harm to BAS networks and to avoid inconveniencing the building occupants. Second, as part of the responsible disclosure policy, we have notified all affected parties about our findings, and they have acknowledged our findings. At the time of writing, six of the bugs have already been patched by the vendors.

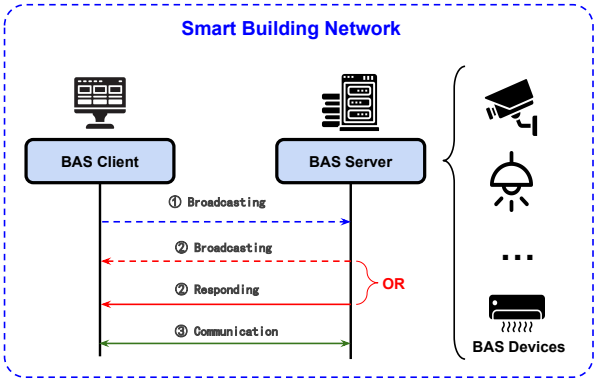

Figure 1: Communication overview between a BAS client, a BAS server, and various BAS devices in a smart building network.

Code Availability: The source code of BASE is available at: https://github.com/AnSECer/BASE.

2 Background

In this section, we present an overview of BAS (§2.1), followed by a brief introduction of the KNX and BACnet communication protocols (§2.2).

2.1 Building Automation Systems (BAS)

In cyber-physical systems (CPS), Building Automation System (BAS) controls and monitors building components such as HVAC, lighting, security scanning, and more. A BAS network has an administrator that manages it via dedicated software. The software is usually proprietary and licensed [3, 18, 41]. To enable communication with a breadth of devices across a potentially large physical area, it is common for BAS protocols to support IP-layer communication (e.g., KNX [2] and BACnet [11]) and form a distributed network. Devices without IP support may connect via physical wiring.

Figure 1 provides an overview of a typical BAS network topology. In this setup, the BAS network comprises a workstation running management or automation software, referred to as the *BAS client*. The BAS client connects to controllers within the network, designated as *BAS servers*, and these servers, in turn, link to the sensors and actuators in the field, known as *BAS devices* (e.g., light, HVAC). The process begins with the BAS client attempting to "discover" the BAS server through broadcast or multicast messages across the network (**Step** ①). The server can announce its presence either by broadcast, multicast, or unicast, i.e., replying directly to the BAS client (**Step** ②). Afterward, the BAS server and BAS client typically communicate directly via unicast, with the BAS client acting as a network client and the BAS server acting as a network server (**Step** ③).

2.2 BAS Communication Protocols

KNX. KNX is a popular building automation protocol, administered by the KNX Association, widely used in Europe and Asia [2]. KNX devices may offer a number of services and expose different variables and functions, which are collectively called "datapoints." Some datapoints manifest as object properties, which may be device- or application-specific. A collection of properties is called an "object". Objects and properties are indexed. For example, `Object 0` is for device-specific properties such as manufacturer ID, accessible via individual addresses. The client can use software such as ETS [3] to connect to the server. As shown in Figure 3, the process involved in the data flow of a tunneling connection is intricate and multifaceted, necessitating a carefully orchestrated exchange of packets, which is detailed as follows:

① The client first opens a tunnel connection to the server (`Tunnel ConnectReq`) and receives a response with the status of the connection (`Tunnel ConnectResp`).
② The client requests to connect to the underlying transport layer of the server (`TunnelReq L_Data.Req (Connect)`). The server acknowledges the tunnel request (`TunnelAck`) and confirms the connect request (`TunnelReq L_Data.Con (Connect)`).
③ The client acknowledges the server's confirmation (`TunnelAck`).
④ The client sends a data request via APCI (`TunnelReq L_Data.Req (<APCI>)`). The server acknowledges (`TunnelAck`) and confirms (`TunnelReq L_Data.Con (<APCI>)`) the APCI request.
⑤ The client acknowledges the server's confirmation (`TunnelAck`). The server sends a transport layer acknowledgement (`TunnelReq L_Data.Ind (ACK)`).
⑥ The client acknowledges the transport layer acknowledgement (`TunnelAck`). The server finally sends the response to the APCI request (`TunnelReq L_Data.Ind (<APCI>)`).
⑦ The client acknowledges receipt of the data (`TunnelAck`). Then the client sends a transport layer acknowledgement request (`TunnelReq L_Data.Req (ACK)`). The server acknowledges (`TunnelAck`) and confirms (`TunnelReq L_Data.Con (ACK)`) the request.
⑧ The client acknowledges the confirmation (`TunnelAck`).
⑨ The client requests to disconnect from the underlying transport layer of the server (`TunnelReq L_Data.Req(Disconnect)`). The server acknowledges (`TunnelAck`) and confirms (`TunnelReq L_Data.Con (Disconnect)`) the request.
⑩ Finally, the client requests to close the tunnel connection (`DisconnectReq`). The server responds with the status of the close request (`DisconnectResp`).

BACnet. BACnet, popular in the US and Canada, was developed by *ASHRAE* [11]. A BACnet server organizes device data into structures called "objects", and each object may have a list of characteristics called "properties". BACnet servers provide services to interact with their objects. Clients (other BACnet devices) use these services to access and manipulate the data within BACnet servers. For example, in a BAS server of the temperature sensors, each sensor is represented as an "Analog Input" object with properties such as "Present Value" (current temperature) and "Object Name." The clients can use the "Read Property" service to retrieve the current temperature from a specific sensor. For instance, the client sends a "Read Property" service request to "Analog Input" object (representing a sensor) with the "Present Value" property, and the BAS server responds with the temperature reading (e.g., 18.5 C).

3 Threat Model and Challenges

3.1 Threat Model

Assumptions. Our aim is to create a specialized fuzzing tool for BAS. This tool assumes that users, including building security analysts, can reasonably access the BAS network (wired or wireless), gather valid IP packet sessions, and send BAS packets. This assumption applies to both fuzzing tests and potential attacks. This assumption is conspicuously sound when applied to fuzzing tests since those conducting these assessments typically include building managers and security analysts. However, we posit that it remains a reasonable premise even in the context of potential attacks. Within any given building, numerous individuals enter and exit daily, and any one of them could conceivably gain access to the building's WiFi network, which often connects to the BAS server. Please note that some devices may require additional authentication or encryption. For such devices, we do not assume our attack will be effective. To target these devices, our discovered exploits would need to be combined with other advanced hacking techniques for unauthorized access (e.g., [6, 7, 10, 13, 19, 33]). For example, BACnet Secure Connection (BACnet/SC) uses TLS and a PKI system for security. Compromising these would require exploiting vulnerabilities in TLS or its implementation within these protocols. An insider attacker, such as someone who gains access to the building, could potentially plant malware and send packets from authenticated devices.

Scope. BAS firmware images are typically proprietary and not readily accessible online. This poses a significant challenge for testers who wish to obtain these firmware images for advanced testing techniques such as firmware rehosting. Due to these constraints, our research has concentrated on blackbox fuzzing, a method that does not require access to the internal workings of the firmware. Blackbox fuzzing is advantageous as it is broadly applicable and can be used to test systems without needing detailed knowledge of their internal architecture.

3.2 Challenges

While fuzzing can be an effective method for discovering bugs quickly and automatically, fuzzing BAS systems presents several major challenges (C) that must be addressed. We have split these challenges into three separate categories: *protocol challenges* (C1), which arise from the protocol irrespective of the specific smart building software or hardware; *client-side challenges* (C2), which arise from the clients used to control the devices; and *server-side challenges* (C3), which arise from the BAS servers.

(C1) Complex BAS Message Structure and Dependencies. Protocols are often structurally complex, as packets typically encompass multiple network layers. For instance, BACnet/IP packets, typically transmitted via UDP, include information about the BACnet network within the UDP payloads, abstracted from the IP network. A simplistic mutation fuzzer might inadvertently corrupt BACnet network-related information (e.g., destination address), potentially prompting the IP-layer destination host to respond with an error message or disregard the request entirely. Such an approach would likely generate numerous futile fuzzy requests. Additionally, packets may contain context-sensitive metadata, such as special counters

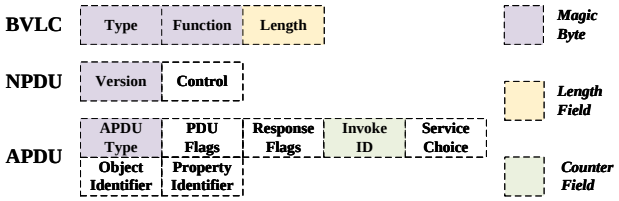

Figure 2: A sample BACnet property read request.

and length fields. Figure 2 illustrates a BACnet request containing magic bytes, length fields, and counter fields. A fuzzer must generate valid values for these fields; otherwise, the entire packet could be discarded. Also, without knowing the correct sequence, a fuzzer might not reach deep code within the target. Figure 3 illustrates this, where a KNX client communicates with a KNX server via an intermediate IP interface. To send fuzzy APCI data consistently, the fuzzer must open the tunnel connection (Steps ①–③), send the necessary acknowledgment frames, and terminate the connection properly (Steps ⑨–⑩). Failure to perform any of these actions will likely cause the server to terminate the connection preemptively.

(C2) Complicated Code Coverage for BAS Clients. Fuzzing smart building clients is challenging because most clients are closed-source and proprietary, making compile-time instrumentation for coverage-guided fuzzing difficult. To make matters worse, the BAS client usually runs an on-demand UDP client as necessitated by the user, e.g., to perform a liveness check on the network or to send a command to a server. Most off-the-shelf fuzzing frameworks cannot directly work with these kinds of applications, as they cannot determine the code coverages of the BAS clients (which is closely related to the functionality of the client). For instance, many fuzzers adopt the "file-parsing" approach popularized by AFL, where the fuzzer assumes the program will terminate after processing the input, and its coverage function is based on this assumption. Other fuzzers such as `WinAFL` and `honggfuzz` can fuzz user-selected functions within long-lived applications by calling the function repeatedly and mutating its input parameters; however, the selected function must either be self-contained, or the user must provide "cleanup" code to reset the application to the fuzzable state, which is non-trivial for proprietary (and complex) programs.

(C3) Closed-source BAS Server and Limited BAS Throughput. Fuzzing BAS applications faces challenges due to the restrictive hardware of BAS servers. Recent fuzzing works on embedded systems emphasize firmware rehosting via emulation software (e.g., QEMU) for analysis or instrumentation [22–24, 42, 49]. However, BAS firmware images are usually proprietary and not available online. Another server-specific challenge concerns the timing of input delivery. BAS devices such as Programmable Logic Controllers (PLCs) "scan" for inputs at fixed intervals, which limits the fuzzing throughput [42]. For example, if inputs are generated too quickly, the device may ignore a portion of them because the scan cycle is not ready. However, as shown in Figure 1, a client does not typically communicate directly with the BAS devices; instead, the BAS

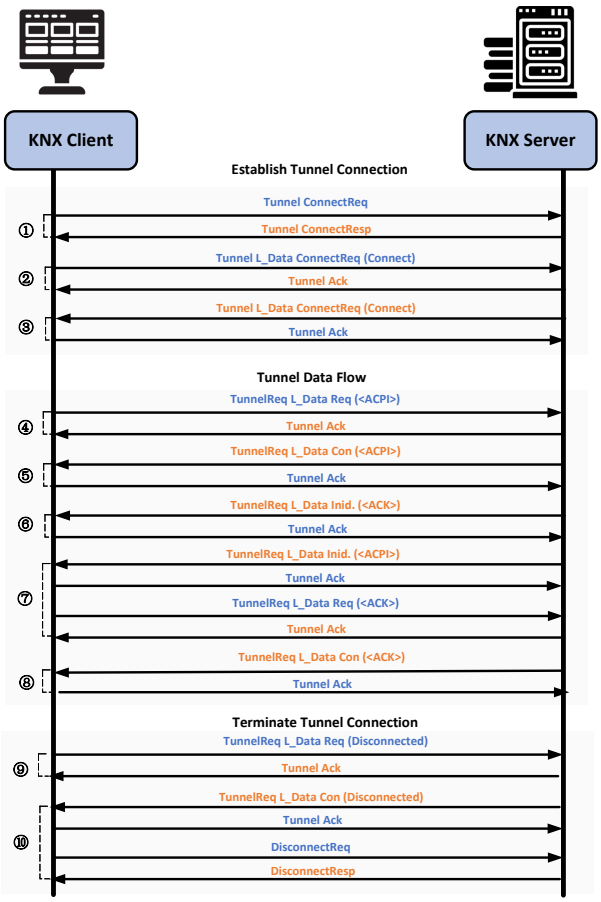

Figure 3: KNX flowchart for a tunneling connection.

server may act on behalf of the devices and forward the appropriate telegrams between the client and the devices. The IP interface of the BAS server may not necessarily be limited by the scan cycle interval imposed on the devices; however, the physical interface between the server and devices is still limited, so the server may not receive data if the client generates them too quickly.

4 BASE Design

In this section, we describe our proposed method for designing a BAS fuzzer. Our tool called BASE, is illustrated by Figure 4 and comprises four primary modules: protocol analyzer, client inspector, server examiner, and core fuzzer. We now explain each module in greater detail:

- **Protocol Analyzer (§4.1).** This module examines the request packets transmitted from the BAS client to the BAS server, as well as those from BAS server to the BAS devices. It scrutinizes and categorizes the individual bytes within each packet based on the BAS server's response to altered requests. The primary purpose of this module is to furnish the fuzzer with the necessary information to generate valid fuzzing requests.
- **Core Fuzzer (§4.2):** This module consolidates data obtained from the protocol analyzer and employs it to conduct fuzzing on the target BAS server or client. Simultaneously, it captures responses from the server or software information from the client, which are subsequently forwarded to the Client Inspector and Server Examiner for the purpose of understanding their code coverage.
- **Client Inspector (§4.3):** This module dynamically instruments with the BAS client software framework to examine response packets from a BAS interface and capture runtime coverage data. It is responsible for detecting client-side crashes and contributing to the overall client-side final results.
- **Server Examiner (§4.4):** Similar to the client inspector, this module directly interrogates the BAS server to retrieve responses and scrutinizes response packets with the aim of assessing the server's fuzzing coverage. Additionally, it manages throughput to ensure that all sessions are successfully conveyed to the BAS server. Its primary role is to identify server-side crashes and make a substantial contribution to the overall server-side final results.

4.1 Protocol Analyzer

We now describe the protocol analyzer, which collects and annotates BAS session data for further fuzzing. As discussed earlier, a BAS protocol can be difficult to fuzz due to the complexity of its frame syntax. For example, a packet may contain magic bytes, context-sensitive data such as length and counter fields, or require a sequence of packets to maintain a specific order. Therefore, the purpose of this module is to identify which fields and packet sequences are context-sensitive and determine the values they are sensitive to. At a high level, the protocol analyzer first collects the packets (*Step I*), then statically analyzes and dynamically probes the BAS system to identify the fields of interest (*Step II*), and finally resolves the dependencies (*Step III*).

Step I: Collecting the Session Data. BASE first collects a corpus consisting of one or more sessions by monitoring the traffic between the server and the client. A *session* in this context refers to a self-contained sequence of packets that can be repeatedly sent to the BAS server with consistent responses. This property is crucial for probing packets, as it involves mutating packets byte-by-byte and comparing the responses to the original. Sessions are generally easy to capture by interacting with BAS frameworks. For instance, ETS can be configured to communicate over the tunnel connection with a KNX interface in the network, capturing all necessary packets for repeatable tunnel connections. For each packet, BASE records the tuple (e.g., timestamp, IP source, port source, IP destination, port destination, raw payload). After capturing a session, BASE further splits it into smaller sub-sessions. A *sub-session* is a continuous sequence of requests followed by a continuous sequence of responses; an illustration is provided by Figure 5. Later on, when BASE discovers which session sequences should preserve their order, the corresponding sub-sessions will contain pointers to each other.

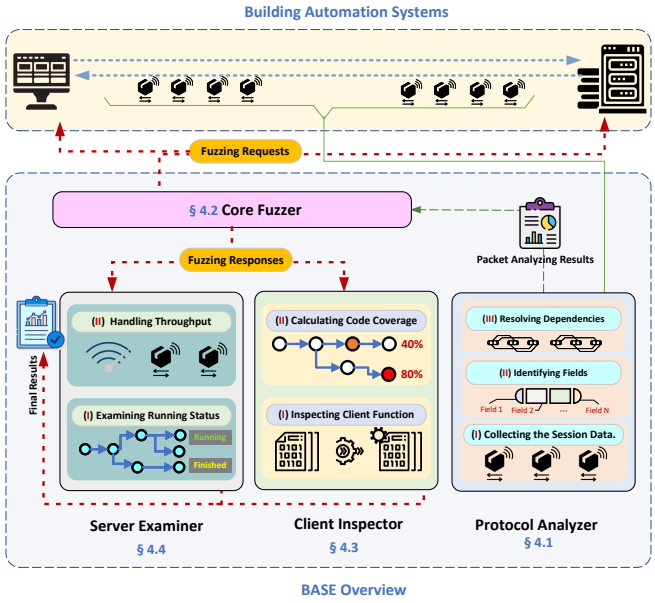

Figure 4: BASE high-level overview.

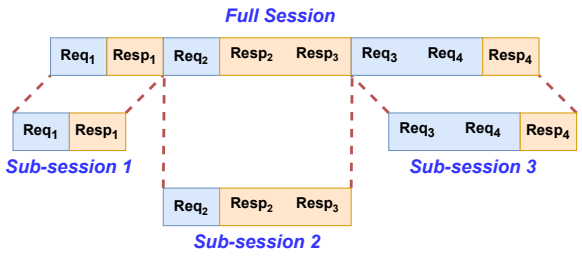

Figure 5: BASE splits sessions into smaller sub-sessions consisting of discrete request-response pairs

Step II: Identifying Fields. The primary idea of this step involves generating distinct packets with specific fields set as either fixed or dynamically changing. These crafted packets are then sent to the server, and its responses are collected. By comparing the fields across various packets, we can determine the type of each field. These identified fields are subsequently utilized in our ensuing fuzzing process. Specifically, we focus on the following fields:

- **Magic Byte.** BASE first probes the session by identifying magic bytes within each packet. A "magic byte" is defined as a byte for which its initially designated value is the sole valid entry. These bytes should remain unchanged during the fuzzing process. To identify these bytes, we iterate over the entire packet, generating multiple new copies of the original packet, each identical except for one randomized byte. These new packets are then sent to the BAS server, along with any other necessary (unmodified) requests in the session. If the BAS server provides an unexpected response or no response at all, and the response is identical for all copies, the byte is initially annotated as a magic byte.
- **Length Field.** A length field is a field whose value depends on the length of the packet. To identify length fields, BASE inserts a single byte into the packet at an index that was not previously marked as magic. If the response differs from the expected response, we inspect the immediately preceding magic field as a length field candidate. The intuition is that length fields might have been erroneously marked as magic since the incorrect length would generate errors. For the bits under consideration, we increment their field value by 1 (rolling over to 0 if necessary) and resend the packet. If the packet results in a new response, it may indicate a new error message. To confirm the validity of the length field, we mutate the value of the injected byte multiple times. If the BAS server responds with a consistent error response not seen before, the field under test is likely a magic byte, and we save the responses for further reference. Conversely, if we observe a response previously associated with a mutated magic byte (i.e., an error response), we can confirm that the field under test is not a length field. If the response matches the expected valid response, we can confirm that the field under test is a length field.
- **Counter Field.** A *counter field* is a field whose value increments from packet to packet. Identifying these fields is crucial since repeated calls to the same request may fail if the counter field is not updated correctly. To identify counter fields in a given packet, BASE employs two criteria. First, in consecutive requests, the counter field should increment, and after reaching its maximum value (e.g., `0xff`), it should reset to the minimum value (e.g., `0x00`). Second, the counter fields in corresponding pairs of requests and responses should match. Our intuition is that true application-specific information, which can vary significantly between packets, generally occurs *after* the counter field, while relevant header information, which tends to be more consistent, generally occurs *before* the counter field. We do not check length fields and other non-magic bytes since they may also vary between packets. However, since length fields can cause offset differences, we adjust our packet search to align with the appropriate offset when a length field is detected. After identifying a matching packet, we compare the values of the selected bytes between packets. If the difference is 1 (or if the preceding packet is `0xff` and the succeeding packet is `0x00`), we consider the field a counter field candidate. To validate, we check the value in a third packet. If the field is absent, we adjust the counter and send the packet to the BAS server. A match confirms the counter field.
- **Passive Field.** *Passive bytes* are defined as bytes where every value is both valid and identical; the BAS server's response is always as expected, regardless of the field's value. These fields can be difficult to identify through static protocol analysis [26] because their behavior is usually target-dependent. For instance, a passive byte may occur if a BAS target failed to implement a certain error-handling mechanism. Nonetheless, annotating passive bytes is important to avoid wasting time fuzzing them. Passive byte identification is straightforward and can be performed concurrently with magic byte

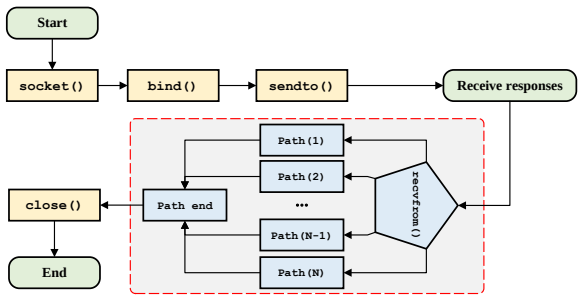

Figure 6: Instrumentation control flow graph for BAS clients running a UDP server.

identification. We generate multiple packets with mutated bytes at a selected offset and send them to the BAS server. If each response matches the expected response, we mark the selected byte as passive.

Step III: Resolving Dependencies. The next challenge is to identify immutable session sequences, such as the KNX tunneling connection. BASE identifies immutable sequences by swapping each adjacent pair of sub-sessions, patching counter fields as necessary, and monitoring the response from the BAS server when each new session is sent out. If a response returns a known error or unexpected response, then we annotate the latter sub-session as a successor to the former by adding a *"next"* pointer from the preceding sub-session to the successive sub-session, as well as a *"prev"* pointer in the opposite direction; in this way, BASE knows to always generate the successive sub-session after the former. If the server returns the same expected response, then the order of this sub-session pair does not matter and we do not have to annotate them.

4.2 Core Fuzzer

The core fuzzer implements the fuzzing component of BASE. We prioritize fuzzing of regular fields, since they are less likely to result in complete semantic or syntax bugs, allowing us to discover new behaviors in the target more quickly: magic bytes and passive bytes have low probabilities of being mutated since our probing modules failed to discover any interesting new behavior in those bytes. While still low, the probability of mutating a context-sensitive byte is kept higher than magic and passive bytes, since we suspect context-sensitive bytes to have more than one valid value. As for length fields, there is a reasonable probability that BASE perturbs the length of the field, either by deleting or inserting new bytes, and immediately patching the length field value to reflect the new byte count, and a slightly smaller probability that the length field value is *not* patched. For cases when the total number of bytes exceeds the maximum size of the length field, we simply set the length field to the maximum value. Furthermore, there is a slim possibility that BASE will insert significantly more bytes than normal into the packet, potentially triggering a buffer overflow or DoS attack. Finally, since BASE must retroactively set the values of the counter fields after it selects the session for fuzzing, there is a minor probability that it will set incorrect values.

Specifically, in a cycle, the fuzzer picks a random sub-session without a "prev" pointer (indicating sequence dependence) and mutates its packets based on the annotations, adjusting timestamps and relevant data as needed. For each packet in the sub-session and for each byte `B` in a packet, the following mutation rules shall apply:

- If `B` is not annotated, then it has a probability `N` to be mutated.
- If `B` is a magic byte, then it has a probability $\frac{N}{100}$ to be mutated.
- If `B` is a passive byte, then it has a probability $\frac{N}{50}$ to be mutated.
- If `B` is part of a length field and the value of the field is `L`, then: (i) There is a probability $\frac{N}{3}$ that we randomly insert or delete up to `L` bytes immediately after `B`. (ii) There is a probability $\frac{N}{100}$ that we insert up to `L * 100` bytes after `B`. (iii) There is a probability $\frac{N}{5}$ that the value of `L` is not patched after the new bytes are inserted or deleted.
- if `B` is part of a counter field, then it has a probability $\frac{N}{40}$ to be set to an invalid value.

4.3 Client Inspector

We present the client inspector of BASE. This module probes response packets to gain insight into the client's functionality. As discussed in (**C2**), fuzzing smart building clients is complex owing to their closed-source nature, and complicated code coverage. As such, this module needs to inspect the functionality of a given BAS client (*Step I*) and calculate the client's code coverage (*Step II*).

Step I: Inspecting Client Function. Considering the source code of the client is unavailable, a method to inspect its functionality is by altering the server's response and monitoring the resultant behavior. However, directly mutating the response packet and forwarding it to the BAS client might not yield a response, as the client may become unresponsive. Our approach involves dynamically instrumenting the BAS client, allowing us to observe its reaction to modified responses. Although the client is closed-source, it still relies on standard system functions, which remain un-obfuscated regardless of the client's source code status.

Specifically, BASE must determine which code regions of the BAS client to instrument, focusing only on the code that processes responses from the BAS server. This can be achieved using a *coverage flag*: when activated, it instruments the target code; when deactivated, the code runs without instrumentation. Since the BAS client communicates with the BAS server over IP (generally UDP), we can use network-specific functions and system calls to identify when the client is processing the response. A BAS client typically runs both a UDP server (for multicast/broadcast data) and a UDP client (for unicast data), each requiring different tracing methods. Below, we provide two examples to illustrate the concept. For simplicity, we use Linux system calls as an example, but the approach applies to Windows using the Winsock API.

- **UDP Server.** As shown in Figure 6, the UDP server starts by opening a socket with the `socket` system call. We instrument the `bind` call to monitor relevant network functions. If the received address matches the address of interest i.e., the broadcast or multicast address, we log the socket. The client then uses `sendto` to interact with the BAS server. In our packet probing approach, BASE operates a decoy BAS server that sends back a modified packet. When this is received, the `recvfrom` call provides details about the socket and response content. If the socket aligns with the `bind`'s, we activate the coverage flag and begin instrumentation. However, determining when to conclude the UDP server's instrumentation is complex, as the `close` call for the socket might only occur at the app's closure. One method is to identify the server's "listening" section awaiting the `recvfrom` call, instructing the tool to deactivate the coverage flag there. This is labor-intensive for each target. An alternative, adopted by BASE, is to auto-deactivate the coverage after a predetermined duration, set by analyzing timestamps in the session corpus to gauge the time between a client's response and the next request. This respects the BAS devices' input synchronization and ensures timely coverage cessation.
- **UDP Client.** We now discuss how to trace a UDP client. Like the server, the client first opens a socket via the `socket` system call. Although a UDP client does not make the `bind` system call, address information for the recipient host can be obtained via the `recvfrom` system call. In this case, we monitor the system call and toggle the coverage flag on when the recipient address matches the BAS server address, also recording the socket value. Eventually, the application closes the connection via the `close` system call, at which point we toggle the coverage flag off and share the coverage information.

Step II: Calculating Code Coverage. We now describe the code coverage calculation algorithm. Our algorithm is inspired by AFL's branch coverage algorithm. AFL calculates coverage by keeping track of which code paths have been executed during the fuzzing process. It uses a bitmap structure to record this information. Each bit in the bitmap corresponds to a specific code path, and when an input takes a particular code path, the corresponding bit is set. When a new path is explored, the coverage score increases accordingly, thus reflecting the overall extent of coverage achieved.

In the original AFL, at each executed branch `i -> j`, for a parent function `i` and a child function `j`, AFL calculates an index $i \oplus (j >> 1)$ and increments `covmap[index]`, where `map` is a 64 kB block of memory shared with the parent fuzzer. The exact values of `i` and `j` are generated randomly at compile-time. AFL defines several hit count "buckets" of the following values: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, and 128 and above. If `covmap[index]` falls within a bucket that was not previously observed, then AFL considers it to be new coverage.

Our code coverage algorithm preserves some of this behavior, with notable modifications. Instead of sharing a memory block with BASE, the instrumentation engine maintains a private coverage map (an array of integers) and calculates a running coverage score based on the values in this map. For each branch `i -> j`, BASE calculates the index value exactly like AFL. The values of `i` and `j` can either be provided directly by the dynamic instrumentation engine or referenced from the entry addresses of the respective functions. When the value of `covmap[index]` reaches a bucket, BASE increments the total coverage score by `primes[index]`, where `primes` is a list of consecutive prime integers. The buckets are mostly identical, except we split the bucket 32-127 into two buckets: 32-63 and 64-127. When the coverage flag is toggled off, only the final coverage score is shared with BASE, allowing it to efficiently compare the coverage score with previous scores.

We note that this method introduces a slight risk of *coverage score collision*, in which the algorithm may generate erroneous coverage score duplicates even if the true coverage was unique. However, due to the `primes` map usage, the risk of collisions is kept minimal. The first time an edge is hit, the score increments by a prime number, guaranteeing that the score could only increment by that amount if that particular edge was hit. For hit counts greater than one, there is a slight chance of duplication. For instance, if the target hits `primes[N] = 13` twice, the coverage score increases by 26. However, a duplication can occur if the target later hits `primes[N] = 5` once and `primes[M] = 7` four times (triggering the third bucket). Still, we observe that the majority of edges only execute once, which keeps the chances of duplication small. To confirm our theory, we wrote a simulation tool that runs 5000 iterations of our code coverage algorithm, with a map size of 10000 and a branch count of 50000. Not a single iteration resulted in a duplicate coverage score. Moreover, duplicates can be avoided entirely by mapping hit counts and indices to `primes[N * S + index]`, where `N` is the `N'th` bucket hit by this index, and `S` is the coverage map size (i.e., size of `covmap`). Then every bucket of every index is guaranteed to always increment the coverage score by a unique value. The downside is that the size of `primes` increases nine-fold, since there are nine buckets.

4.4 Server Examiner

We introduce the server examiner component of BASE. This module dispatches fuzz payloads to the server. As highlighted in (C3), fuzzing servers present challenges due to their closed-source nature and restricted throughput. Consequently, this module is tasked with determining the server's operational status (*Step I*) and managing its limited throughput (*Step II*).

Step I: Examining Server Running Status. For effective fuzzing of a target BAS device, understanding and monitoring the fuzzing status is crucial. This often involves emulating and instrumenting server-side firmware, which is typically inaccessible. However, the primary goal of such emulation and instrumentation is to gain code coverage insights. If alternative side channels can provide server-side code coverage data, then firmware access becomes unnecessary. Therefore, BASE can pose as a BAS client, sending fuzzy requests to the BAS server and observing responses that suggest new coverage areas.

To target a specific device, the user should first collect a session corpus of packets addressing the device in their requests. When BASE probes the session, it will likely mark the address-relevant

bytes as magic bytes, since invalid addresses return error responses (or not return any responses at all). When receiving a response, BASE will compare it to previous responses to determine if it is unique, excluding any frequently changing counter fields. If the response is unique, the fuzzed session is added to the corpus. The "prev" pointer is preserved if present, but the "next" pointer is deleted since the next session depends on the original unfuzzed session. Lastly, BASE checks the hardware's liveness by sending periodic *heartbeat monitors*, i.e., requests with guaranteed and predictable responses. A suitable heartbeat monitor can be arbitrarily selected from any sub-session sequence in our corpus, as we have already confirmed the consistency of those sessions.

Step II: Handling Throughput. As discussed in (C3), timing synchronization is crucial because BAS devices scan for inputs at regular intervals, and requests may be lost if sent too frequently. Therefore, during the network monitoring phase, we collect timestamp information for each packet. Later, when we probe and annotate the session, we preserve the timing by sending requests at the same rate as observed. When listening for responses from the server, we wait up to twice the original response time to account for unexpected network delays. By replicating the original session timing as closely as possible, we avoid input synchronization issues.

Due to the forced input synchronization, the packet probing module can take a long time if the session contains many packets or if the packets themselves are large. To increase the module's overall performance, when BASE begins probing a new packet, it first refers back to previously analyzed packets and compares the magic or passive bytes, length fields, and counter fields. If the packet under test appears to contain the same fields as the previous packets, we annotate those fields without sending requests to the BAS server and waiting for responses. This approach minimizes the amortized time cost of the packet probing module.

5 Evaluation

5.1 Experimental Setup

BASE Running Environment: All experiments were conducted on a Dell XPS 15 9510 laptop with Intel Core i9-11900H CPU and 32 GB of RAM. The majority of experiments were performed on Windows 11 directly on the host, while some Linux-specific experiments were performed in an Ubuntu 22.04 virtual machine. We used the Python library `Scapy` to monitor traffic between BAS servers and clients and seed our session corpus. For dynamic instrumentation, we used `Intel Pin`, which supports Windows and Linux targets on 32- and 64-bit architectures.

BAS Server-side Targets: In total, we targeted 7 KNX devices and 4 BACnet devices. The KNX devices are QAW912, KNX RF/TP Coupler 673 Secure, KNX IP LineMaster 762, 5WG1 258-2DB12, EIKON 21840, KNX Virtual, and the GDB181.1E/KN. The BACnet devices are PMDTBXB, BASRT-B, HNDTA2BX, and GH2SMBBR1. These devices cover a variety of smart building functions including room heating control, particulate matter (PM) sensing, temperature, and humidity sensing, and so forth. Table 1 presents a summary of all 11 devices. For our experiments, the BASRT-B by Contemporary Controls serves as the BACnet/IP interface (BAS server), while the KNX IP LineMaster 762 by Weinzierl serves as the KNXnet/IP interface. KNX Virtual, a Windows application by the KNX Association, is a virtual interface, while also implementing 27 virtual KNX devices such as actuators, alarm modules, room controllers, and more; we do not include those devices in our discussion since we did not evaluate them individually.

BAS Client-side Targets: As shown in Table 2, we performed our experiments on 3 KNX software clients and 3 BACnet software clients. The KNX applications were ETS, knxd, and Calimero. The BACnet applications were Innea BACnet Explorer, YABE, and CAS BACnet Explorer. Of these, ETS, Innea BACnet Explorer, CAS BACnet Explorer, and YABE are proprietary Windows applications. knxd is an open-source library that runs as a daemon on Linux hosts. Calimero is an open-source Java library.

5.2 Discovered Vulnerabilities

BASE successfully discovered 13 new BAS vulnerabilities, including 8 client-side vulnerabilities and 5 server-side vulnerabilities. Table 3 summarizes our findings. All client-side vulnerabilities result in nearly immediate termination of the application. Three of the server-side bugs resulted in denial-of-service. In the case of BASRT-B, a full power cycle is necessary to resume access to the BACnet/IP interface. For the 5WG1 (presence detector), the device appeared to have permanently lost its functionality and attempts to reprogram the device using ETS and restore it failed. The vulnerabilities discovered in the KNX IP LineMaster result in a temporary denial-of-service in which configuration requests from the client are completely ignored for several seconds; eventually, the LineMaster terminates the connection with the client and resumes normal operation.

Client-Side Vulnerabilities: We begin by focusing on client-side vulnerabilities (V). In the context of these vulnerabilities, we consider two potential scenarios: the attacker can either be the malicious server or an attacker who compromises the gateway. In both cases, the attacker gains the capability to send carefully crafted packets to the victim clients without restraint. To be more specific:

- **(V1) Innea BACnet Explorer:** Innea BACnet Explorer can crash if a malicious BAS server or a gateway sends a fuzzy `I-Am` packet to the application; this payload is regularly used to respond to a BACnet `Who-Is` discovery request. The crash occurs due to a memory access violation (error code `0xc0000005`).
- **(V2) CAS BACnet Explorer:** CAS BACnet Explorer can become unresponsive indefinitely if a malicious BAS server sends a fuzzy `readProperty` acknowledgement packet containing a vendor proprietary object type, without specifying the property ID. This payload allows BACnet devices to respond to requests to read property information; typically the device shall respond either with an error code or the property contents.
- **(V3) knxd:** knxd can crash if the daemon is started with the `-listen-tcp` option, which exposes an IP server on port 6720 for remote KNX devices to communicate. The bug results in process aborts to corrupted addresses in the KNX payload.

Name	Manufacturer	Protocol	Description
QAW912	Siemens	KNX RF	Heat controller
PMDTBXB	Greystone	BACnet MSTP	PM sensor
KNX RF/TP Coupler 673 Secure	Weinzierl	KNX RF, KNX TP	KNX RF/TP coupler
KNX IP LineMaster 762	Weinzierl	KNXnet/IP, KNX TP	KNX Interface
BASRT-B	Contemporary Controls	BACnet/IP, BACnet TP, Ethernet	BACnet Interface
HNDTA2BX	Greystone	BACnet MSTP	Duct humidity/temperature sensor
5WG1 258-2DB12	Siemens	KNX TP	Presence detector
EIKON 21840	VIMAR	KNX TP	4-button programmable switch
GH2SMBBR1	Greystone	BACnet MSTP	Temperature, humidity and CO_2 sensor
KNX Virtual	The KNX Association	KNXnet/IP	Various virtual devices
GDB181.1E/KN	Siemens	KNX TP	VAV compact controller (damper)

Table 1: Summary of BAS servers that were tested.

Name	Developer	Protocol	Platform
ETS	The KNX Association	KNX	
knxd	Matthias Urlichs	KNX	
Calimero	Calimero Project	KNX	
Innea BACnetExplorer	Inneasoft	BACnet	
YABE	Morten Kvistgaard	BACnet	
CAS BACnet Explorer	Chipkin AutomationSystems	BACnet	

Table 2: Summary of BAS client software frameworks that were tested (Windows, MacOS, and Linux).

Name	Protocol	Type	Error Summary
(V1) Innea BACnetExplorer	BACnet		Memory access violation
(V2) CAS BACnet Explorer	BACnet		Unresponsive
(V3) knxd	KNX		Abort #1
(V4) knxd	KNX		Abort #2
(V5) knxd	KNX		Segmentation fault
(V6) Calimero	KNX		Out of memory
(V7) ETS	KNX		Out of memory
(V8) KNX Virtual	KNX		Index out of bounds
(V9) LineMaster	KNX		Unresponsive
(V10) Presence Detector	KNX		Permanent brick
(V11) BASRT-B	BACnet		Crash #1
(V12) BASRT-B	BACnet		Crash #2
(V13) GDB181.1E/KN	KNX		Unresponsive

Table 3: Summary of vulnerabilities. : Server, and : Client.

(V4) knxd: Similarly, this bug also causes knxd crash when the daemon is started with the `-listen-tcp` option. This bug results in process aborts due to failed assertion checks. It fails when certain packets do not include the *Transport Layer Protocol Data Unit* (TPDU) data structure, which carries information about service requests and responses.

(V5) knxd: This bug exhibits a comparable trigger condition (i.e., when BASE transmits fuzzy KNX packets to the server via port 6720). It causes a segmentation fault.

(V6) Calimero: When running the KNXnet/IP server implemented by Calimero, a Java exception `java.lang.OutOfMemoryError` can occur when a malicious BAS client sends a KNX request with service code `0xffff`. Before crashing, the software will rapidly and repeatedly echo an error message, and the memory consumption of the process will gradually increase until the exception occurs.

(V7) KNX Virtual: KNX Virtual can crash if a malicious client sends a truncated KNX request that is missing the "Total Length" field, leading to a .NET `System.IndexOutOfRangeException`.

(V8) ETS: By sending a corrupted `Search Response Extended` packet to ETS after it sends a `Search Request`, the software quickly begins to consume a large amount of memory, leading to resource exhaustion and performance degradation of the software and host system. The payload contains a corrupted data information block.

Server-Side Vulnerabilities: We are now directing our attention towards server-side vulnerabilities (V). In contrast to client-side attacks, exploiting server-side vulnerabilities merely necessitates the attacker's connection to a server, making it a more practical scenario in real-world situations. To be more specific:

(V9) LineMaster: By opening a KNX configuration connection with the LineMaster and repeatedly sending Configuration Request messages, the device will eventually stop responding to the client, even for completely valid requests. In normal circumstances for the KNX `management` service, a Configuration Request (for a valid connection) shall always be met with a Configuration Acknowledgement by the KNX server; this is analogous to the Tunnel Requests and Tunnel Acknowledgements illustrated in Figure 3. However, by spamming Configuration Requests, the LineMaster eventually stops communicating with the client and eventually terminates the connection.

(V10) Presence Detector: The 5WG1 258-2DB12 (presence detector) can become unresponsive by sending fuzzy routing indication packets. KNX `routing` services may carry the same application-layer payloads as the tunneling services, but there is no acknowledgment requirement as opposed to management or tunneling services. Our experiments caused the presence detector to become completely unusable. Typically, a KNX device can become "reprogrammed" by using ETS to download the application to the device.

(V11) **BASRT-B:** We discovered one bug in the BASRT-B "BASrouter" interface. It occurs when the malicious client broadcasts a BACnet request with a corrupted APDU field. The interface becomes completely unresponsive until receiving a full power cycle.

(V12) This bug has also been identified in the BASRT-B "BASrouter" interface. It manifests when an `Abort` message is sent, causing the interface to become entirely unresponsive until it undergoes a complete power cycle.

(V13) **KNX Damper:** The GDB181.1E/KN VAV compact controller ("KNX damper") can become temporarily unresponsive due to fuzzy `Routing Indication` packets. If the packet contains a corrupted APDU field, the damper will become unresponsive for a few seconds. The attacker can send the fuzzy packet in an infinite loop to disable the damper indefinitely.

5.3 Performance Evaluation

To understand its performance, we evaluated our BASE in terms of code coverage and compare it with state-of-the-art fuzzers.

Code Coverage. In our study, we tested the code coverage performance of knxd, which is a KNX gateway software program. The rationale for selecting this particular software is knxd is identified as containing the highest number of vulnerabilities within its category (3 out of 8 vulnerabilities). Testing on the software with a high vulnerability density allows us to more effectively demonstrate the coverage capabilities of our fuzzing tool. Unlike traditional fuzzers, which may crash and terminate testing when a vulnerability is detected, our fuzzer can continue fuzzing even if a vulnerability is triggered, showcasing its robustness and thoroughness. knxd [14] is also very popular in BAS. Its widespread adoption enhances the relevance and applicability of our findings to real-world scenarios.

The fuzzing process was carried out continuously for 24 hours, during which our fuzzer monitored the execution of knxd and logged coverage data. Specifically, it tracked the transitions between basic blocks (edges) in the program's control flow graph and measured the number of executions per second to assess performance. Figure 7 and Figure 8 illustrate the results, showing that our fuzzer discovered a total of 17,616 unique edges in knxd's control flow graph. Additionally, the fuzzer maintained an average speed of 138.91 executions per second, demonstrating efficient input processing and sustained performance.

Comparison with Other Fuzzers. We chose to compare our fuzzer with Boofuzz [5]. Boofuzz is an open-source fuzzing framework that provides various tools and functionalities to help users define, execute, and monitor fuzzing tests. Its highly customizable nature allows for the addition of extra implementations to support packet analysis and message field dependency resolution. We tested Boofuzz using its default packet resolution and fuzzing policies. In some of our tests, we enhanced Boofuzz's capabilities by integrating BASE's packet analysis capabilities. To enable dependency support in Boofuzz, we would need to re-implement its feedback mechanism.

The comparison is conducted in the following way: if Boofuzz can identify the vulnerability in significantly less time or discover a new vulnerability that our fuzzer cannot find, then Boofuzz is deemed

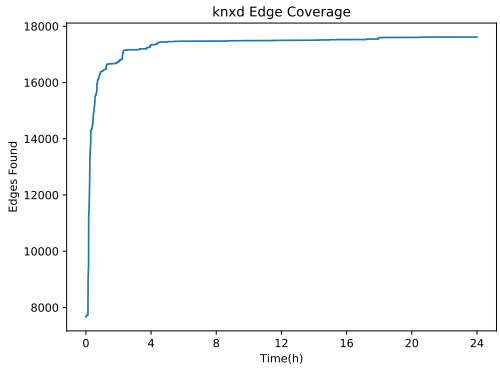

Figure 7: Code coverage for *knxd*

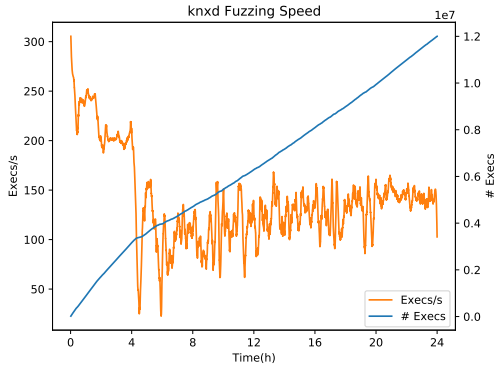

Figure 8: Fuzzing Speed for *knxd*

superior. Otherwise, our fuzzer is considered better. To test Boofuzz, we designed two experiments. In the first experiment, Boofuzz is configured with zero knowledge, meaning all inputs are randomly generated without considering the structure of the messages. In the second experiment, we assume that Boofuzz knows the format of the message (e.g., Boofuzz already knows the message consists of a string and two integers). This is achieved by using our BASE packet message analysis. A notable feature of Boofuzz is the "`fullrange`" configuration option. When set to `fullrange=True`, Boofuzz tests every possible value within the specified range for a given field, rather than testing only a subset or a random sampling of the values. However, with `fullrange=True`, Boofuzz systematically tests all values. In our experiment, we toggled the fullrange setting on to test a specific field thoroughly. This allowed us to compare the performance and thoroughness of Boofuzz and our fuzzer under different configurations and knowledge levels. All the experiments were executed three times to ensure reliability.

Table 4 shows the results and the following observations are made. First, after 24 hours of continuous operation, Boofuzz, without any prior knowledge, sent a total of 2,883,793 (on average) mutated messages but did not find any vulnerabilities. This indicates that completely random fuzzing without targeted strategies can be extremely inefficient and may fail to uncover vulnerabilities

	1st Attempt		2nd Attempt		3rd Attempt		Average
	F	pkt #	F	pkt #	F	pkt #	
BooFuzz (Random)	✗	2,856,753	✗	2,650,042	✗	2,994,584	2,883,793
BooFuzz (w/ `fullrange`)	✗	3,059,973	✗	2,838,229	✗	2,690,164	2,862,788
BASE	✓	50,217	✓	41,612	✓	91,524	61,117

Table 4: Comparison with Boofuzz. "F" represents "found".

in the system. Second, when Boofuzz has knowledge of the packet structure, it did not find vulnerabilities after sending 2,862,788 (on average) mutated messages. This inefficiency of Boofuzz can be attributed to its control of mutation testing depth through the `max_depth` parameter. This is a useful feature as it provides flexibility and customizability to the testing process. However, in practical application, this approach can lead to a significant amount of redundant testing. For example, suppose there are three bit fields, *A*, *B*, and *C*. Different `max_depth` settings result in single-field mutations, combinations of two fields, and combinations of three fields. As `max_depth` increases, the number of test cases grows exponentially, but many cases are redundant. For instance, the combinations *A=0, B=1, C=0* and *B=1, A=0, C=0* are treated as different test cases even though they are essentially the same input. This redundancy not only wastes resources but also reduces the efficiency of the testing process. Finally, our fuzzer finds the vulnerability with an average of 61,117 mutated messages. This is because it not only recognizes the structure of messages but also resolves the relationships between fields, enhancing its effectiveness in discovering vulnerabilities.

6 Attack Case Studies

In this section, we discuss the attack case studies to demonstrate the consequences caused by crashes. We do not include more sophisticated attacks beyond crashes or unresponsive BAS devices because our focus is on using fuzzing to understand the BAS system security landscape and test software and devices for vulnerabilities, rather than designing more sophisticated attacks. Fuzzing research (e.g., [8]) often targets availability-related bugs. The discovered crashes may be caused by various reasons such as buffer overflow and may be further exploited, leading to advanced attacks, which can be our future work. Please also note that in all our tested devices, none of the BAS devices support additional authentication (e.g., BACnet/SC). This means that we can initiate the DoS attack without employing any additional hacking techniques.

Attacks against BACnet Router: We now present a detailed case study of our attack against the BASrouter controller. Figure 9 illustrates our attack testbed. The BASrouter is wired via MSTP to the GH2SMBBR1 temperature/humidity sensor. A power supply connects to both the BASrouter and sensor. The BASrouter exposes the BACnet/IP interface at `192.168.92.68:47808`. To communicate with and monitor the devices, a victim's laptop connects to an Ethernet switch, which is connected to the BASrouter; in practice, the BASrouter may instead be connected to a wireless router, and the attack can be performed remotely. The victim runs the Innea BACnet Explorer software to monitor the devices. To detect devices, BACnet Explorer sends `Who-Is` broadcast requests on the network and listens for any `I-Am` responses. The BASrouter also exposes a

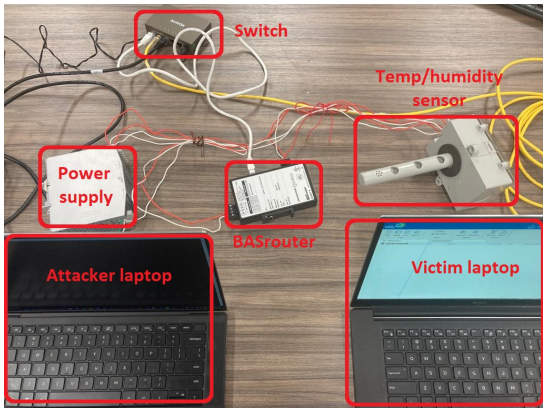

Figure 9: Our testbed for the attack on the BASRT-B.

web server, which the victim can visit to configure the router and monitor network information. Meanwhile, the attacker's laptop also connects to the BACnet/IP interface using the Ethernet switch.

To conduct the attack, the attacker laptop just sends a crafted BACnet/IP packet onto the network. This packet contains a corrupted APDU payload, `ff1083`. After the BASrouter receives the packet, it becomes completely unresponsive, and the victim laptop can no longer monitor either device on the network. The web server also becomes inaccessible. Although the fault exists solely in the BASrouter, the temperature/humidity sensor also becomes inaccessible because it depends on the BASrouter to relay traffic to the user or other devices. In our experiments, a full power cycle was required in order to restore the normal functionality. In a practical BAS application, this attack could potentially disable communication to dozens of equipment and have serious consequences (e.g., loss of climate control, fire systems failure).

Attacks against KNX Damper: This attack works against the GDB181.1E/KN VAV compact controller, which we refer to as the KNX damper for simplicity. Figure 10 illustrates our testbed. The damper is connected to the KNX LineMaster router via twisted pair (KNX TP). The LineMaster is connected to the power supply on the lower right in the figure. Meanwhile, the damper must be powered using a secondary power supply (in the upper right of the figure) and a transformer. Similar to the BASrouter example, the victim's laptop can communicate with the KNX network by connecting to an Ethernet switch, which connects to the LineMaster. The LineMaster exposes the KNXnet/IP interface at `169.254.123.252:3671`, while the multicast interface can be reached at `224.0.23.12:3671`. The victim's laptop runs the ETS software to monitor the LineMaster and damper. ETS provides a feature called "Individual Address check", which attempts to establish a Tunnelling connection and read some basic information from the device. The victim can use this feature to monitor the damper's liveness. The attacker also connects to the LineMaster via the Ethernet switch.

To conduct the attack, the attacker sends a number of KNXnet/IP packets to the multicast interface. The payload contains a `Routing Indication` packet to the LineMaster with the destination address 0.2.249, which is the programmed address of the damper. Therefore,

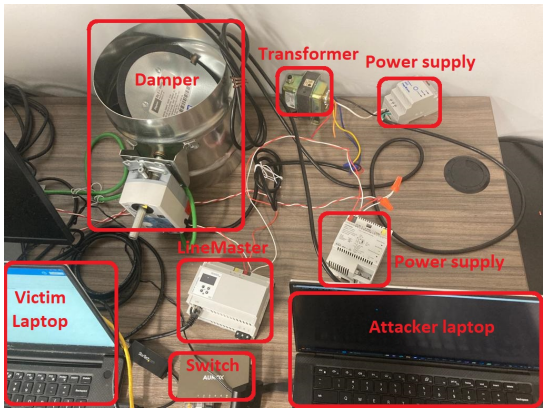

Figure 10: Our testbed for the attack on the KNX damper.

the LineMaster forwards the payload to the damper. The packet contains a corrupted APCI `PropValueWrite` payload, attempting to write data to the device at an invalid property index (2817). After the damper receives this payload, it becomes unresponsive on both KNX interfaces for a few moments. For instance, if the victim tries to use ETS to perform the Individual Address Check, the address will appear to be inactive. In our experiments, the attacker only needs to send the payload to the LineMaster about once per 2 seconds to render the damper completely inaccessible.

Attacks against ETS: This attack works against the ETS software framework. For the setup, the attacker just needs to obtain a copy of ETS, which is available for free as a demo. We confirmed this bug on ETS version 6.0.6, which is the newest version available at the time of writing. The attacker runs ETS and the attack script on the same machine. Once ETS opens, it periodically tries to identify KNX routers on the network by sending out `Search Request` multicast packets on different interfaces.

To conduct the attack, the attacker listens on an interface and waits for ETS to send the discovery request. It then responds with the number of our crafted packets, containing a `Search Response Extended` packet with a corrupted data information block. After the attacker sends this payload, ETS begins to gradually consume more of the system's memory, until the software eventually becomes unresponsive. The bug may cause Windows to prompt the user that the software is not responding. It's worth mentioning that ETS is considered the "official" smart building automation software for KNX, as it is developed and maintained by the KNX Association, which also maintains the KNX standard itself. Therefore, any bugs that impact ETS can have wide repercussions on the whole KNX community.

7 Related Work

This section summarizes the state-of-the-art works related to general smart building security, particularly those aimed at BACnet and KNX, as well as fuzzing techniques applied to hardware (e.g., embedded systems and IoT devices) and software. Our work is novel in three major aspects as highlighted in our contribution list. First, we are the first to systematically assess the system security of BAS networks. Second, we evaluated BASE on 11 BAS servers and identified 13 new vulnerabilities. Third, we conducted attack case studies to demonstrate the implications of these vulnerabilities, which may pose serious threats such as delayed fire detection, loss of climate control, and security breaches. It is worth noting that although context-aware fuzzing tools can handle complicated protocol states, they cannot be directly used to fuzz BAS, as their default policies are not optimized for BAS (as discussed in §5.3).

BAS Security. BAS security research [17] has predominantly focused on the network layer, addressing threats such as key management flaws [15], denial-of-service [33], impersonation attacks [6], and data theft (e.g., [4, 6, 7, 9, 10, 13, 16, 19–21, 31, 33, 37, 44, 46]). This emphasis arises from the inherent security gaps in popular smart building protocols (e.g., BACnet [11], KNX [2]), which lack client/server authentication. Optional security extensions, e.g., BACnet Secure Connect [12] KNX IP Secure [28], and KNX Data Secure [19, 27], remain underutilized. For example, W. Granzer and W. Kastner observed that many BAS implementations adopt the flawed "security by obscurity" approach, and they investigate flaws in popular protocols such as ZigBee, LonTalk, KNX, and Z-Wave [15]. They found that these protocols are commonly vulnerable to data availability attacks and scalability issues, while key management services are commonly centralized, introducing a single point of failure. P. Ciholas et al. analyzed 45 papers on smart building security and concluded that most works focus on defense techniques against BACnet, KNX, and Lon [10]. They further explored BAS in terms of the field layer (BAS devices) and automation/management layer (BAS server), showing that most works focus on the latter. In contrast, BAS software security has been largely overlooked. In [43], the authors used a basic fuzzing script testing one KNX device. Our work targets software attacks across the entire BAS network, emphasizing the imperative need to scrutinize BAS server and client through a software security lens, as evident in our assessment and preliminary results.

Fuzzing Research. Many fuzzing works emphasize assessing the security of embedded systems, notably IoT devices [8, 22, 23, 50]. For example, FIRM-COV enhances augmented emulation by incorporating a panic handler for system-mode emulation [22]. ICSFuzz targets programmable logic controllers in ICS control applications, writing arbitrary data to the device by intercepting input control calls [42]. In the realm of software fuzzing, recent advancements have primarily tackled two key challenges: *application complexity* and *input complexity* [25, 30, 35, 39]. For example, ZAFL enhances dynamic instrumentation performance through binary transformations [30]. VUzzer utilizes static and dynamic analyses to inform the fuzzer about application data and control flows [35]. There are also a number of efforts that deal with understanding and handling structurally complex inputs (e.g., [32, 34, 45, 48]). For example, Wang et al. [45] developed Taintscope, which uses dynamic analysis to identify checksum checks in the application. Different from those efforts, we aim for greater flexibility and universality in smart building security assessment.

Park et al. [32] introduce a new technique called aspect-preserving mutation. This technique aims to stochastically preserve desirable properties, termed aspects, during the mutation process. The focus is on maintaining structure and type preservation. However, their

techniques are designed for JavaScript, and they do not deal with the dependency relationship between the different field of a fuzzing message. FreeDom [47], a cluster-friendly DOM fuzzer supporting both generative and coverage-guided modes, uses a context-aware intermediate representation to describe HTML documents with accurate data dependencies. While very powerful, FreeDom currently does not support parsing BAS protocols. To process BAS protocols, FreeDom would need to convert BAS messages to its custom intermediate representation, FD-IR, which requires additional effort. PeriScope [40] focuses on the interface between hardware and the operating system, which is critical in many IoT devices. It identifies and fuzzes interfaces where vulnerabilities may exist. FIRM-AFL [50] extends the AFL fuzzer to handle IoT firmware by using process emulation to execute firmware code on a host machine. It augments traditional fuzzing with hardware-specific feedback. However, such efforts would require hardware knowledge or the proper emulation of the firmware image. While not exclusively IoT-focused, Nyx [38] offers insights into fuzzing environments where hypervisors are used, which can include advanced IoT devices. It leverages fast snapshots and affine types to enhance fuzzing efficiency. However, it is subject to complexity of setup and applicability to specific use cases involving hypervisors. REDQUEEN [1] improves fuzzing by understanding the correspondence between input bytes and program state, which can be particularly useful for protocol fuzzing in IoT devices. However, it is subject to complexities in establishing and maintaining input-to-state mappings.

8 Conclusion

In this work, we present BASE, which assesses the security of BAS networks through fuzzing. BASE automates the process of identifying protocol structures, dynamically instruments clients for comprehensive code coverage analysis, and monitors responses to discover new coverage areas. Additionally, it utilizes collected timestamps to optimize server input scan intervals, thereby handling throughput limits. Through extensive evaluations conducted on various BAS servers and clients, BASE has identified and reported 13 previously undisclosed vulnerabilities. These findings are not purely theoretical; they have real-world security implications for BAS systems, as confirmed by our case studies. We anticipate that in the near future, more security assessment tools like BASE will be introduced to enhance the protection of the integrity and reliability of BAS networks.

9 Acknowledgments

We thank the shepherd and the anonymous reviewers for their valuable suggestions and comments. This research was supported in part by Drexel Startup Fund, by US National Science Foundation (NSF) Awards 1931871 and 2325451, by Jiangsu Provincial Key R&D Programs Grant Nos. BE2021729, BE2022680, and BE2022065-5, Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant No. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.
[2] KNX Association. Knx. https://www.knx.org/, 2023.
[3] KNX Association. What is ets professional? https://www.knx.org/knx-en/for-professionals/software/ets-professional/, 2023.
[4] Rage Usha Bhargavi. Smart home automation and security using raspberry module.
[5] Boofuzz Documentation. boofuzz: Network protocol fuzzing for humans. https://boofuzz.readthedocs.io, 2024. Accessed: 2024-08-01.
[6] Michael Cash, Christopher Morales, Shan Wang, Xipeng Jin, Alex Parlato, Qun Zhou Sun, and Xinwen Fu. On false data injection attack against building automation systems. *arXiv preprint arXiv:2208.02733*, 2022.
[7] Michael Cash, Shan Wang, Bryan Pearson, Qun Zhou, and Xinwen Fu. On automating bacnet device discovery and property identification. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021.
[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.
[9] Woo-Hyun Choi and Jung-Ho Lewe. Advancing fault detection in building automation systems through deep learning. *Buildings*, 14(1):271, 2024.
[10] Pierre Ciholas, Aidan Lennie, Parvin Sadigova, and Jose M Such. The security of smart buildings: a systematic literature review. *arXiv preprint arXiv:1901.05837*, 2019.
[11] BACnet Committee. Ashrae bacnet. https://bacnet.org/, 2023.
[12] BACnet Committee. Bacnet secure connect. https://bacnetinternational.org/bacnetsc/, 2023.
[13] Behrang Fouladi and Sahand Ghanoun. Security evaluation of the z-wave wireless protocol. *Black hat USA*, 24:1–2, 2013.
[14] GitHub. Github - knxd/knxd. https://github.com/knxd/knxd, 2024. Accessed: 2024-08-01.
[15] Wolfgang Granzer and Wolfgang Kastner. Security analysis of open building automation systems. In *Computer Safety, Reliability, and Security: 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings 29*, pages 303–316. Springer, 2010.
[16] Adib Habbal, Mohamed Khalif Ali, and Mustafa Ali Abuzaraida. Artificial intelligence trust, risk and security management (ai trism): Frameworks, applications, challenges and future research directions. *Expert Systems with Applications*, 240:122442, 2024.
[17] David G Holmberg and D Evans. *BACnet wide area network security threat assessment*. US Department of Commerce, National Institute of Standards and Technology, 2003.
[18] Inneasoft. Bacnet explorer. https://inneasoft.com/en/bacnet-explorer/, 2023.
[19] Aljosha Judmayer, Lukas Krammer, and Wolfgang Kastner. On the security of security extensions for ip-based knx networks. In *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*, pages 1–10. IEEE, 2014.
[20] Ben Kereopa-Yorke. Building resilient smes: Harnessing large language models for cyber security in australia. *Journal of AI, Robotics & Workplace Automation*, 3(1):15–27, 2024.
[21] Haena Kim, Yejun Kim, and Seungjoo Kim. A study on the security requirements analysis to build a zero trust-based remote work environment. *arXiv preprint arXiv:2401.03675*, 2024.
[22] Juhwan Kim, Jihyeon Yu, Hyunwook Kim, Fayozbek Rustamov, and Joobeom Yun. Firm-cov: High-coverage greybox fuzzing for iot firmware via optimized process emulation. *IEEE Access*, 9:101627–101642, 2021.
[23] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual computer security applications conference*, pages 733–745, 2020.
[24] Chongqing Lei, Zhen Ling, Yue Zhang, Yan Yang, Junzhou Luo, and Xinwen Fu. A friend's eye is a good mirror: Synthesizing {MCU} peripheral models from peripheral drivers. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7085–7102, 2024.
[25] Kaizhen Liu, Ming Yang, Zhen Ling, Yue Zhang, Chongqing Lei, Junzhou Luo, and Xinwen Fu. RIoTFuzzer: Companion App Assisted Remote Fuzzing for Detecting Vulnerabilities in IoT Devices. In *Proceedings of the 31th Conference on Computer and Communications Security (CCS'24)*, 2024.
[26] Kaizheng Liu, Ming Yang, Zhen Ling, Huaiyu Yan, Yue Zhang, Xinwen Fu, and Wei Zhao. On manually reverse engineering communication protocols of linux-based iot systems. *IEEE Internet of Things Journal*, 8(8):6815–6827, 2020.
[27] Vassilios Lourdas. Knx data secure. https://support.knx.org/hc/en-us/articles/360012689639-KNX-Data-Secure, March 2020.
[28] Vassilios Lourdas. Knx ip secure. https://support.knx.org/hc/en-us/articles/360012666599-KNX-IP-Secure, March 2020.

[29] Lan Luo, Yue Zhang, Bryan Pearson, Zhen Ling, Haofei Yu, and Xinwen Fu. On the security and data integrity of low-cost sensor networks for air quality monitoring. *Sensors*, 18(12):4451, 2018.
[30] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium*, 2021.
[31] Andrzej Ożadowicz. Generic iot for smart buildings and field-level automation—challenges, threats, approaches, and solutions. *Computers*, 13(2):45, 2024.
[32] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.
[33] Matthew Peacock, Michael N Johnstone, Craig Valli, O Camp, P Mori, and S Furnell. Security issues with bacnet value handling. In *ICISSP*, pages 546–552, 2017.
[34] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.
[36] ReportLinker. Building automation systems market - growth, trends, covid-19 impact, and forecasts (2022 - 2027). https://www.reportlinker.com/p06360537/, October 2022.
[37] Francesco Rosati. *Enhancing Security in Smart Buildings: Traffic Classification for Automated Access Control.* PhD thesis, Politecnico di Torino, 2024.
[38] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2597–2614, 2021.
[39] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 166–180, 2022.
[40] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.
[41] Chipkin Automation Systems. Cas bacnet explorer. https://store.chipkin.com/products/tools/cas-bacnet-explorer, 2023.
[42] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. Icsfuzz: Manipulating i/os and repurposing binary code to enable instrumented fuzzing in ics control applications. In *USENIX Security Symposium*, pages 2847–2862, 2021.
[43] Claire Vacherot. Sneak into buildings with knxnet/ip. In *Sneak into buildings with KNXnet/IP*, 2020.
[44] Markus Voggenreiter, Florian Angermeir, Fabiola Moyón, Ulrich Schöpp, and Pierre Bonvin. Automated security findings management: A case study in industrial devops. *arXiv preprint arXiv:2401.06602*, 2024.
[45] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.
[46] Yonghao Wang, Yao Wang, Zhenqin Yang, Qingshan Wang, Hao Zhang, et al. A hybrid building information modeling and collaboration platform for automation system in smart construction. *Alexandria Engineering Journal*, 88:80–90, 2024.
[47] Wen Xu, Soyeon Park, and Taesoo Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.
[48] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE symposium on security and privacy (SP)*, pages 769–786. IEEE, 2019.
[49] Yue Zhang, Melih Sirlanci, Ruoyu "Fish" Wang, and Zhiqiang Lin. When Compiler Optimizations Meet Symbolic Execution: An Empirical Study. In *Proceedings of the 31th Conference on Computer and Communications Security (CCS'24)*, 2024.
[50] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium*, pages 1099–1114, 2019.