SEVENTH FRAMEWORK PROGRAMME

Information & Communication Technologies
Trustworthy ICT

NETWORK OF EXCELLENCE

A European Network of Excellence in Managing Threats and Vulnerabilities in the Future Internet: *Europe for the World* [†]

Deliverable D6.4: Final Report on Smart Environments

Abstract: This deliverable presents a number of research results produced in the *SysSec* project, relevant to the Smart Environment WorkPackage and in relation to the Research Roadmap.

Contractual Date of Delivery	September 2014
Actual Date of Delivery	October 2014
Deliverable Dissemination Level	Public
Editor	Magnus Almgren
Contributors	All *SysSec* partners
Quality Assurance	IICT-BAS, POLIMI

The *SysSec* consortium consists of:

FORTH-ICS	Coordinator	Greece
Politecnico Di Milano	Principal Contractor	Italy
Vrije Universiteit Amsterdam	Principal Contractor	The Netherlands
Institut Eurécom	Principal Contractor	France
IICT-BAS	Principal Contractor	Bulgaria
Technical University of Vienna	Principal Contractor	Austria
Chalmers University	Principal Contractor	Sweden
TUBITAK-BILGEM	Principal Contractor	Turkey

[†] The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 257007.

Document Revisions & Quality Assurance

Internal Reviewers

1. Vladimir Dimitrov (IICT-BAS)
2. Vincenzo Gulisano (Chalmers)
3. Marina Papatriantafilou (Chalmers)
4. Philippas Tsigas (Chalmers)
5. Stefano Zanero (POLIMI)

Revisions

Ver.	Date	By	Overview
0.0.11	29/10/2014	#2	Final review of deliverable consistency.
0.0.10	28/10/2014	#4	Further updates by internal review.
0.0.9	27/10/2014	#5	Second review by QMC on deliverable concerning format, style, and content.
0.0.8	24/10/2014	#1	Thorough review of QMC for all chapters.
0.0.7	20/10/2014	#3	Detailed review of introduction, conclusions, and roadmap relations.
0.0.6	24/9/2014	Editor	Consistency check from all partners.
0.0.5	23/9/2014	#4	Feedback on introduction.
0.0.4	23/9/2014	Editor	Quality check by all chapter contributors.
0.0.3	4/9/2014	#2	Merging of packages, references, macros, etc.
0.0.2	15/7/2014	Editor	Chapters committed by partners.
0.0.1	28/5/2014	Editor	Outline and preliminary article selection complete.
0.0.0	10/2/2014	Editor	First outline of document.

Contents

List of Figures

Foreword

One of the objectives of the *SysSec* Network of Excellence is to strengthen the system security research in Europe. In the *Smart Environment* workpackage, we consider the security of networks and devices that comprise smart environments. With previous deliverables, we have given an overview of the field. The objective of this fourth and final deliverable is to demonstrate ongoing research.

The deliverable is a compilation of important results obtained within the *SysSec* Network of Excellence. We have chosen to include the research results, in the actual form of published articles, based on two criteria. First, we consider their relation to the research defined in the Red Book [178]. Second, the objective is to let the compilation as a whole paint a picture of the diversity of the research needed for smart environments. The chapters illustrate important steps for smart environments: analysis of software, vulnerability discovery and the resulting system hardening, detection of attacks as a complement to hardening techniques, and privacy implications.

Each chapter is also introduced with a preamble to relate the research to the roadmaps produced by *SysSec*. Together, the chapter selection and the relationship to the roadmap will help established researchers, but also new PhD students in system security, to be more involved in the European research scene.

The result of the smart environment workpackage is the four-volume set of deliverables. The first three volumes describe the research efforts for the focal points of the work package: the sensor network, the smart car, and the smart grid. Thus, they serve as an introduction to the field by giving a succinct but useful summary of the state of the art. This fourth deliverable completes the set by investigating a set of advanced and more complex research questions.

Previous deliverables in this series

In the first deliverable, *Report on The State of the Art in Security in Sensor Networks*, we considered low-capability devices such as sensor nodes and their respective networks. Research-wise, we considered the fundamental network-service algorithms for such environments.

In the second deliverable, *Intermediate Report on the Security of The Connected Car*, we considered a specific application area to focus the discussion. The *connected car*, as a research area, is being developed actively both by industry and in academia and with reported security problems.

In the third deliverable, *Advanced Report on Smart Environments*, we focused on the smart grid, the common term to refer to the new, evolving future grids with adaptive functionality. We gave an overview of this complex domain, suitable for researchers and students in computer science, and then highlighted security-relevant issues. We surveyed the SCADA and AMI, discussed privacy issues, as well as showing related work in building intrusion detection systems for this environment.

1 Introduction

1.1 Background

The objective of the *SysSec* Network of Excellence is to strengthen the system security community in Europe in three dimensions. Firstly, we develop and collect material for teaching system security to students. Secondly, we produce a yearly roadmap of threats and important research areas; an effort that culminated with *the Red Book* [178]. The objectives of the Red Book is for policy makers to understand future research needs but also to guide PhD students searching for a viable thesis topic in system security. Thirdly, in addition to these activities, the members of *SysSec* also conduct research in system security topics aligned to the defined roadmap and publish the results in established international conferences and journals. The research is structured along three topics (workpackages).

In the *Smart Environment* workpackage, we especially consider the security of networks and devices that comprise smart environments. Sensors are becoming ubiquitous; they exist all around us and are able to collect data from the environment, calculate or aggregate values, and then transmit them elsewhere. People and companies speak about ubiquitous computing, pervasive computing and the Internet of things as having a profound impact on the future where everything will be connected. The challenge is to take these concepts, where the range and composition of what they mean vary from person to person, into concrete ideas and topics useful for the European system security community.

1.2 Focus of the workpackage

We define the focus of the smart environment work package to be low-capability devices, such as simple sensor networks, but also example environments to demonstrate more complex concepts with heterogeneous systems with more capable hardware (a system of systems). As low-capability devices grow in both sophistication and market penetration, users start to use them for a wide variety of purposes,

thus trusting these devices by collecting and storing vital private information on them even though the devices are not capable of running simple traditional security mechanisms.

Parameters of the individual devices will remain limited in the near future, compared to normal ICT equipment. Such limitations will, in turn, have consequences on the security mechanisms that can be deployed. A good example is cryptographic primitives. These limited devices may not today have the capability to use asymmetric ciphers. Even though they become more capable in the future, one still will have to consider the resource-constrained environment when designing and implementing the security primitives. Actually, the increased security of being able to add one more bit to a node's key might be negated if the adversary can use a regular computer for his attack, as a regular computer will have similar or even better performance gains in the same time frame.

Devices in smart environments are also exposed to many hostile environments, where assumptions made for traditional security solutions no longer hold (as explored in deliverable D6.1). Furthermore, as there is a continuous range of such devices and what they are capable of, a threat and the corresponding mitigating security mechanism may look very different depending on the type of device and the environment it is located within. In some environments, a single compromised unit might be unacceptable. In others, a few compromised units will not affect the system detrimentally as long as the aggregate data in the whole environment is almost correct. For yet other environments, the two cases are very similar. These environments consist of simple but many very homogeneous units, meaning that if a single one is compromised the attack can easily be repeated to control the whole network. Sometimes, the research community has a clear understanding of the vulnerabilities (e.g. RFID tags), but no solutions for the problems. In other cases, we cannot model the complexity of the environment and we need fundamental research to understand capabilities and limitations. *Thus, one emphasis of the work package is to survey security solutions for very constrained devices such as the sensor network.*

Even though the individual devices of the smart environment are important, also the network and the applications built on top of the infrastructure will play a crucial role for the security of the system. *The second emphasis for the workpackage is thus on example environments*. These environments are complex (system of systems), important for society (critical infrastructures, closely related to human health, etc.), as well as being actively researched and deployed by society as whole. In *SysSec*, we chose early on to especially consider vehicular systems and the smart grid, as these environments are

- in the process of being deployed or modernized,
- very complex, and
- with a need for research efforts to better understand them.

Both the smart car and the electricity grid contain legacy systems or use legacy protocols, which may have a profound effect on the overall security.

1.3 The workpackage deliverables

The overarching guidelines described in the previous section set the scene for the work that has been completed during the *SysSec* project for the smart environment work package. The goal has been to involve more European researchers in topics of interest in system security, and especially smart environments. For that reason, there has been an *educational* component, to help established researchers but also new PhD students in system security to understand the special security challenges faced in smart environments. Seminal results have been described and state of the art research (aligned to the roadmap) has been performed and reported for several important aspects of smart environments.

The result is the four-volume set of deliverables from the smart environment work package. The first three volumes describe the research efforts for the focal points of the work package: the sensor network, the smart car, and the smart grid. These deliverables explain the problems faced in different types of smart environments, what has been done and currently ongoing research efforts in Europe. Thus, they serve as an introduction to the field by giving a succinct but useful summary of the state of the art. In the *Report on the State of the Art in Security in Sensor Networks*, we consider low-capability devices such as sensor nodes and their respective networks. The second deliverable, *Intermediate Report on the Security of The Connected Car*, focuses on the future connected car. The third deliverable, the *Advanced Report on Smart Environments*, describes the idea of the smart grid with a focus on security of some of its domains.

As the previous deliverables give an overview of the field, the emphasis of this fourth deliverable is to dive deep into the research questions that are pursued and to present a selection of the results obtained in the *SysSec* project during the last four years.[1] Even though the *SysSec* members have produced significant results,[2] we have selected to highlight but a few research contributions in this particular deliverable. Simply put, we find that a quality selection of key results produce a more interesting deliverable than what could have been obtained including a majority of the results. We also frame the inclusion of each contribution with a discussion on their relationship to the latest research roadmap. By purpose, we present these contributions in the form of actual research articles to emphasize that they represent research results in the project.[3] *These research results, with the discussion on their relation to the roadmap, will thus serve as examples for European system security researchers on important topics to be tackled in the near future.*

[1] Previous deliverables have also included some research contributions. In D6.2, we discussed *A security layer for automotive services* and *Remote control of smart meters: friend or foe?*.

[2] Please see the project home page for a full list of publications: http://www.syssec-project.eu/publications/

[3] The contributions are in fact preprints of articles that have been published by the *SysSec* team.

1.4 Outline of the volume

The rest of this deliverable is structured along chapters, each representing a research result from *SysSec*. We have chosen to include the research results based on two criteria. First, we consider their relation to the research defined in the Red Book. Second, the objective is to let the collection as a whole paint a picture of the diversity of the research needed for smart environments. The chapters illustrate important steps for smart environments: analysis of software, vulnerability discovery and the resulting system hardening, detection of attacks as a complement to the hardening technique, and privacy implications. We illustrate the parts in Figure 1.1.

Fig. 1.1: The topics of the compilation

The first papers describe methodologies for analysis of firmware, followed by results on correcting or mitigating found problems. As an orthogonal complement, we also need detection of possible attacks in the systems. For that reason, we include results for very efficient measurement validation as well as an outline of a system to detect attacks in one example environment. Finally, we also discuss privacy implications in relation to smart environments.

In Chapter 2, we describe a methodology to dynamically analyze firmwares of embedded devices. We presented an early version in *Advanced Report on Smart Environments*, but we are now able to report more details of the AVATAR research. To emphasize the need for such analysis, we include the results of the first public, large-scale analysis of firmware images in Chapter 3, where over 32.000 firmware images were statically analyzed.

We then turn to one of the problems found in code – the buffer overflow. In Chapter 4, we describe Dowser, a guided fuzzer that can be combined with an analysis technique such as AVATAR to find buffer overflow and underflow vulnerabilities. However, sometimes with legacy systems we cannot fix and recompile firmware. In Chapter 5 we describe a novel technique to protect existing C binaries from memory corruption attacks on both control data and non-control data.

In Chapter 6 and 7 we turn to the problem of detecting attacks and validating data as efficiently as possible with an IDS for the smart grid. Chapter 8 then demonstrates risks to privacy by discussing de-anonymization of large datasets.

1.5 Smart Environment Related Works of the *SysSec* Consortium

In this section, there is a more comprehensive list of works of the *SysSec* project related to the smart environment workpackage. A majority of these are not covered in detail in this deliverable due to the limited space of this report.

- Zhang Fu, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. **Online Temporal-Spatial Analysis for Detection of Critical Events in Cyber-Physical Systems.** 2014 IEEE International Conference on Big Data (IEEE BigData 2014), October 2014, Washington DC, USA.
- Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. **Online and Scalable Data Validation in Advanced Metering Infrastructures.** The 5th IEEE PES Innovative Smart Grid Technologies (ISGT) European 2014 Conference. October 2014, Istanbul, Turkey.
- Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. **METIS: a Two-Tier Intrusion Detection System for Advanced Metering Infrastructures.** 10th International Conference on Security and Privacy in Communication Networks (SecureComm) 2014. September 2014, Beijing, China.
- Farnaz Moradi, Tomas Olovsson, and Philippas Tsigas. **A Local Seed Selection Algorithm for Overlapping Community Detection.** In Proceedings of the 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM14). August 2014, Beijing, China.
- Andrei Costin, Jonas Zaddach, Francillon Francillon, Aurlien, Davide Balzarotti. **A Large Scale Analysis of the Security of Embedded Firmwares.** In Proceedings of the 23rd USENIX Security Symposium (USENIX Security). August 2014, San Diego, CA, USA.
- Vincenzo Gulisano, Magnus Almgren, Marina Papatriantafilou. **POSTER: METIS: a Two-Tier Intrusion Detection System for Advanced Metering Infrastructures.** In Proceedings of the 5th International Conference on Future Energy Systems (ACM e-Energy). June 2014. Cambridge, UK.
- Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. **PROSPECT.** In Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2014). June 2014. Kyoto, Japan.
- Erik Bosman and Herbert Bos. **Framing Signals A Return to Portable Shellcode.** In Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland). May 2014. San Jose, CA, USA.
- Zhang Fu, Olaf Landsiedel, Magnus Almgren, and Marina Papatriantafilou. **Managing your Trees: Insights from a Metropolitan-Scale Low-Power Wireless Network.** In Proceedings of the 3rd Workshop on Communica-

tions and Control for Smart Energy Systems (CCSES), IEEE INFOCOM. April 2014. Westin Harbour Castle Toronto, ON, Canada.

- William Johansson, Martin Svensson, Ulf Larson, Magnus Almgren, and Vincenzo Gulisano. **T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols.** In Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST). April 2014. Cleveland, Ohio, USA.
- Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. **Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.** In Proceedings of the Network and Distributed System Security Symposium (NDSS). February 2014, San Diego, USA.
- Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik Olivier Blass, Aurelien Francillon, Travis Goodspeed, Moitrayee Gupta, Ioannis Koltsidas. **Implementation and Implications of a Stealth Hard-Drive Backdoor.** In Proceedings of the 2013 Annual Computer Security Applications Conference (ACSAC). December 2013, New Orleans, LA, USA.
- Gabriele Bonetti, Marco Viglione, Alessandro Frossi, Federico Maggi Stefano Zanero. **A Comprehensive Black-box Methodology for Testing the Forensic Characteristics of Solid-state Drives.** In Proceedings of the 2013 Annual Computer Security Applications Conference (ACSAC). December 2013, New Orleans, LA, USA.
- Zlatogor. Minchev and Luben Boyanov. **Smart Homes Cyberthreats Identification Based on Interactive Training.** In Proceedings of the 3rd International Conference on Application of Information and Communication Technology and Statistics in Economy an d Education (ICAICTSEE). December 2013, Sofia, Bulgaria.
- Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, Sotiris Ioannidis. **ASIST: Architectural Support for Instruction Set Randomization.** In Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS). November 2013, Berlin, Germany.
- Tudor Valentin, Magnus Almgren, and Marina Papatriantafilou. **Analysis of the impact of data granularity on privacy for the smart grid.** Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society. ACM, 2013.
- Stiliyan Georgiev, Zlatogor Minchev. **An Evolutionary Prototyping for Smart Home Inhabitants Wearable Biomonitoring.** In Proceedings of Conjoint Scientific Seminar Modelling & Control of Information Processes. November 2013, Sofia, Bulgaria.

- Mariano Graziano, Andrea Lanzi, Davide Balzarotti. **Hypervisor Memory Forensics.** In Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID). October 2013, Saint Lucia.
- Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, Herbert Bos. **Dowsing for overflows: A guided fuzzer to find buffer boundary violations.** In Proceedings of the 22nd USENIX Security Symposium (USENIX-SEC). August 2013, Washington, DC, USA.
- Matthias Neugschwandtner, Martina Lindorfer, Christian Platzer. **A view to a kill: Webview exploitation.** In Proceedings of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET). August 2013, Washington, DC, USA.
- Zlatogor Minchev, Luben Boyanov, Stiliyan Georgiev. **Security of Future Smart Homes. Cyber-Physical Threats Identification Perspectives.** In Proceedings of the National Conference with International Participation in Realization of the EU project Development of Tools Needed to Coordinate Intersectorial Power and Transport CIP Activities at a Situation of Multilateral Terrorist Threat. Increase of the Capacity of Key CIP Objects in Bulgaria. June 2013, Sofia, Bulgaria.
- Shlomi Dolev, Omri Liba, Elad M. Schiller. **Self-Stabilizing Byzantine Resilient Topology Discovery and Message Delivery.** In Proceedings of the 2013 International Conference on Networked Systems (NET- SYS). May 2013, Marrakech, Morocco.
- Zhang Fu, Marina Papatriantafilou. **Off The Wall: Lightweight Distributed Filtering to Mitigate Distributed Denial of Service Attacks.** In Proceedings of the31st IEEE International Symposium on Reliable Distributed Systems (SRDS). October 2012, Irvine, CA, USA.
- Andreas Larsson, Philippas Tsigas. **Self-stabilizing (k,r)- Clustering in Clock Rate-Limited Systems.** In proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO). July 2012, Reykjavk, Iceland.
- Asia Slowinska, Traian Stancescu, Herbert Bos. **Body armor for binaries: preventing buffer overflows without recompilation.** In proceedings of the 2012 USENIX Annual Technical Conference (ATC). June 2012, Boston, MA, USA.
- Farnaz Moradi, Tomas Olovsson, Philippas Tsigas. **An Evaluation of Community Detection Algorithms on Large-Scale Email Traffic.** In proceedings of the 11th International Symposium on Experimental Algorithms (SEA). June 2011, Bordeaux, France.
- Zhang Fu, Marina Papatriantafilou, Philippas Tsigas. **Mitigating distributed denial of service attacks in multiparty applications in the presence of**

clock drifts. In IEEE Transactions on Dependable and Secure Computing (TSDC), Volume 9, Issue 3. May 2012.

- Farnaz Moradi, Tomas Olovsson, Philippas Tsigas. **Towards Modeling Legitimate and Unsolicited email Traffic Using Social Network Properties.** In proceedings of the 5th Workshop on Social Network Systems (SNS). April 2012, Bern, Switzerland.
- Mihai Costache, Valentin Tudor, Magnus Almgren, Marina Papatriantafilou, Christopher Saunders. **Remote control of smart meters: friend or foe?** In Proceedings of the 7th European Conference on Computer Network Defense (EC2ND). September 2011, Gteborg, Sweden.
- Andreas Larsson, Philippas Tsigas. **A Self-stabilizing (k,r)-clustering Algorithm with Multiple Paths for Wireless Ad-hoc Networks.** In proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS 2011), June 2011, Minneapolis, Minnesota, USA.
- Pierre Kleberger, Tomas Olovsson, Erland Jonsson. **Security Aspects of the In-Vehicle Network in the Connected Car.** In proceedings of the 2011 IEEE Intelligent Vehicles Symposium (VI 2011). June 2011, Baden-Baden, Germany.
- Farnaz Moradi, Magnus Almgren, Wolfgang John, Tomas Olovsson, Philippas Tsigas. **On Collection of Large-Scale Multi-Purpose Datasets on Internet Backbone Links.** In proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS). April 2011, Salzburg, Austria.
- Zhang Fu, Marina Papatriantafilou, Philippas Tsigas. **CluB: A Cluster Based Proactive Method for Mitigating Distributed Denial of Service Attacks.** In proceedings of the 26th ACM Symposium on Applied Computing (SAC). March 2011, TaiChung, Taiwan.
- Asia Slowinska, Traian Stancescu, Herbert Bos. **Howard: a dynamic excavator for reverse engineering data structures.** In proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS). February 2011, San Diego, CA, USA.
- Andreas Larsson, Philippas Tsigas. **Self-stabilizing (k,r)-Clustering in Wireless Ad-hoc Networks with Multiple Paths.** Brief announcement in Proceedings of 14th International Conference On Principles Of Distributed Systems (OPODIS). December 2010, Tozeur, Tunisia.
- Phuong Nguyen, Wil Kling, Giorgos Georgiadis, Marina Papatriantafilou, Anh Tuan Le, Lina Bertling. **Distributed Routing Algorithms to Manage Power Flow in Agent-Based Active Distribution Network.** Proceedings of 1st Conference on Innovative Smart Grid Technologies Europe. Gteborg, Sweden, October 2010.

2

Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares

Preamble: Relation to the Research Roadmap

The risk that cyber-criminals can tamper with the firmware of embedded devices is mentioned many times in our research roadmaps. For instance, our first roadmap (D4.1) presented a scenario ("The Peccadillo") in which a consumer hacks her own smart meter by installing a custom firmware to reduce her electricity cost. Moreover, the chapters on current and emerging threats in malware and fraud often listed hardware security, and in particular the fact that an attacker can completely reprogram the internal firmware of a device, as one of the most worrying threat.

This trend culminated in the second research roadmap (D4.2), where we explicitly stress the importance of developing firmware analysis tools:

> The availability of open source tools, specifications, and techniques so that researchers can properly analyze the emerging technologies is a key factor for the success of system security in this area.

The following paper presents the first technique to perform advanced dynamic analysis of firmware code:

Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares" Network and Distributed System Security (NDSS) Symposium , San Diego (USA) – February 2014

Abstract

To address the growing concerns about the security of embedded systems, it is important to perform accurate analysis of firmware binaries, even when the source code or the hardware documentation are not available. However, research in this field is hindered by the lack of dedicated tools. For example, dynamic analysis is one of the main foundations of security analysis, e.g., through dynamic taint tracing or symbolic execution. Unlike static analysis, dynamic analysis relies on the ability to execute software in a controlled environment, often an instrumented emulator. However, emulating firmwares of embedded devices requires accurate models of all hardware components used by the system under analysis. Unfortunately, the lack of documentation and the large variety of hardware on the market make this approach infeasible in practice.

In this paper we present Avatar, a framework that enables complex dynamic analysis of embedded devices by orchestrating the execution of an emulator together with the real hardware. We first introduce the basic mechanism to forward I/O accesses from the emulator to the embedded device, and then describe several techniques to improve the system's performance by dynamically optimizing the distribution of code and data between the two environments. Finally, we evaluate our tool by applying it to three different security scenarios, including reverse engineering, vulnerability discovery and hardcoded backdoor detection. To show the flexibility of Avatar, we perform this analysis on three completely different devices: a GSM feature phone, a hard disk bootloader, and a wireless sensor node.

2.1 Introduction

An embedded system consists of a number of interdependent hardware and software components, often designed to interact with a specific environment (e.g., a car, a peacemaker, a television, or an industrial control system). Those components are often based on basic blocks, such as CPUs and bus controllers, which are integrated into a complete custom system. When produced in large quantities, such customization results in a considerable cost reduction. For large quantities, custom built integrated circuits (ASIC) are preferred as they allow to tailor functionality according to the specific needs, which results in cost reduction, better integration, and a reduction of the total number of parts. Such chips, also called System on a Chip (SoC), are often built from a standard CPU core to which both standard and custom hardware blocks are added. Standard blocks, commonly called *IP Cores*, are often in the form of a single component that can be integrated into a more complex design (e.g., memory controllers or standard peripherals). On the other hand, custom hardware blocks are often developed for a specific purpose, device, and manufacturer. For example, a mobile phone modem may contain a custom voice processing DSP, an accelerator for the GSM proprietary hardware cryptography (A5 algorithms) and an off-the-shelf USB controller.

Over the years, such SoCs have significantly grown in complexity. Nowadays, they often include Multiple Processors (MPSoC) and complex, custom, hardware devices. As a consequence, virtually every embedded system relies on a different, application specific, system configuration. As a witness of this phenomenon, the website of ARM Ltd., which provides one of the most common CPU core used in embedded systems, lists about 200 silicon partners[1]. Most of those partners are producing several product families of SoCs relying on ARM cores. This leads to a huge number of systems on the market, which are all different, but all rely on the same CPU core family.

Unfortunately, the increasing pervasiveness and connectivity of embedded devices significantly increased their exposure to attacks and misuses. Such systems are often designed without security in mind. Moreover visible features, low time to market, and reduction of costs are the common driving forces of their engineering teams. As a consequence, an increase in the number of reports of embedded systems exploitation has been recently observed, often with very serious consequences [44, 58, 64, 84, 95, 112, 195, 206, 244, 264]. To make things worse, such systems frequently play an important role in security-relevant scenarios: they are often part of safety critical systems, integrated in home networks, or they are responsible to handle personal user information. Therefore, it is very important to develop the tools and techniques that would make easier to analyze the security of embedded systems.

In the traditional IT world, dynamic analysis systems play a crucial role in many security activities - ranging from malware analysis and reverse engineering, to vulnerability discovery and incident handling. Unfortunately, there is not an equivalent in the embedded system world. If an attacker compromises the firmware of a device (e.g., a smart meter or a PLC in a Stuxnet-like attack scenario [112]) even vendors often do not have the required tools to dynamically analyze the behavior of the malicious code.

Dynamic analysis allows users to overcome many limitations of static analysis (e.g., packed or obfuscated code) and to perform a wide range of more sophisticated examinations [105] - including taint propagation [154, 249], symbolic and concolic execution [53, 72, 92], unpacking [156], malware sandboxing [1, 13], and whitebox fuzzing [122, 123].

Unfortunately, all these techniques and their benefits are still not available in the world of embedded systems. The reason is that in the majority of the cases they require an emulator to execute the code and possibly monitor or alter its execution. However, as we will explain in Section 2.2, the large number of custom and proprietary hardware components make the task of building an accurate emulator a daunting process. If we then consider that additional modules and hardware plugins should be developed for each embedded system on the market, we can easily understand the infeasibility of this approach.

[1] http://www.arm.com/community/partners/silicon.php

In this paper, we present a technique to fill this gap and overcome the limitation of pure firmware emulation. Our tool, named *Avatar*, acts as an orchestration engine between the physical device and an external emulator. By injecting a special software proxy in the embedded device, *Avatar* can execute the firmware instructions inside the emulator while channeling the I/O operations to the physical hardware. Since it is infeasible to perfectly emulate an entire embedded system and it is currently impossible to perform advanced dynamic analysis by running code on the device itself, *Avatar* takes a hybrid approach. It leverages the real hardware to handle I/O operations, but extracts the firmware code from the embedded device and *emulates* it on an external machine.

To summarize, in this paper we make the following contributions:

- We present the design and implementation of *Avatar*, a novel dynamic analysis framework that allows a user to emulate the firmware of an embedded device.
- We discuss several techniques that can be used to optimize the performance of the system and to adapt *Avatar* to the user's needs. We also show how complex dynamic analysis applications (such as concolic execution) can be implemented on top of *Avatar*.
- We evaluate *Avatar* by applying it to three different security scenarios, including reverse engineering, vulnerability discovery, and backdoor detection. To show the flexibility of our system, each test was performed on a completely different class of devices.

2.2 Dynamic Firmware Analysis

While the security analysis of firmwares of embedded devices is still a new and emerging field, several techniques have been proposed in the past to support the debugging and troubleshooting of embedded systems.

Hardware debugging features (mostly built around In-Circuit Emulators [69, 157, 183] and JTAG-based hardware debuggers [11]) are nowadays included in many embedded devices to simplify the debugging procedure. However, the analysis remains extremely challenging and often requires dedicated hardware and a profound knowledge of the system under test. Several debugging interfaces exist, like the Background Debug Mode (BDM) [255] and the ARM CoreSight debug and trace technology [255]. Architecture-independent standards for debugging embedded devices also exist, such as the IEEE NEXUS standard [12]. Most of these technologies allow the user to access, copy, and manipulate the state of the memory and of the CPU core, to insert breakpoints, to single step through the code, and to collect instructions or data traces.

When available, hardware debugging interfaces can be used to perform certain types of dynamic analysis. However, they are often limited in their functionalities

and do not allow the user to perform complex operations, such as taint propagation or symbolic execution. In fact, these advanced dynamic analysis techniques require an instruction set simulator to interpret the firmware of the embedded target. But for a proper emulation of the embedded system, not only the CPU, but all peripheral devices need to be emulated. Without such a support, the emulated firmware would often hang, crash, or in the best case, show a different behavior than on the real hardware. Such deviations can be due, for example, to incorrect memory mappings, active polling on a value that should be changed by the hardware, or the lack of the proper hardware-generated interrupts or DMA operations.

To overcome these problems, researchers and engineers have resolved to three classes of solutions, each with its own limitations and drawbacks:

- *Complete Hardware Emulation*
  Chipounov [70] and Kuznetsov et al. [166] analyze device drivers by relying on an emulated PCI bus and network card that return symbolic values. This approach has the main drawback that it requires to emulate the device properly. While this is not much of a problem for well understood devices, like a PCI network card supported by most PC emulation software, it can be a real challenge in embedded systems and can be just impossible when the hardware is not documented. Unfortunately, lack of documentation is the rule in the embedded world, especially in complex proprietary SoCs.

  In some cases, accurate system emulators are developed as part of the product development to allow the firmware development team to develop software while the final hardware is still not available. However, those emulators are usually unavailable outside the development team and they are often not designed for code instrumentation, making them unable to perform basic security analysis like tainting or symbolic execution.

- *Hardware Over-Approximation*
  Another approach consists in using a generic, approximated, model of the hardware. For example, by assuming interrupts can happen at any time or that reading an IO port can return any value. This approach is easy to implement because it does not require a deep knowledge of the real hardware, but it can clearly lead to false positives, (e.g., values that will never be returned by the real system) or misbehavior of the emulated code (when a particular value is required). This approach is commonly used when analyzing small systems and programs that are typically limited to a few hundreds lines of code, as showed by Schlich [215] and Davidson et al. [92]. However, on larger programs and on complex peripherals this approach will invariably lead to a state explosion that will prevent any useful analysis.

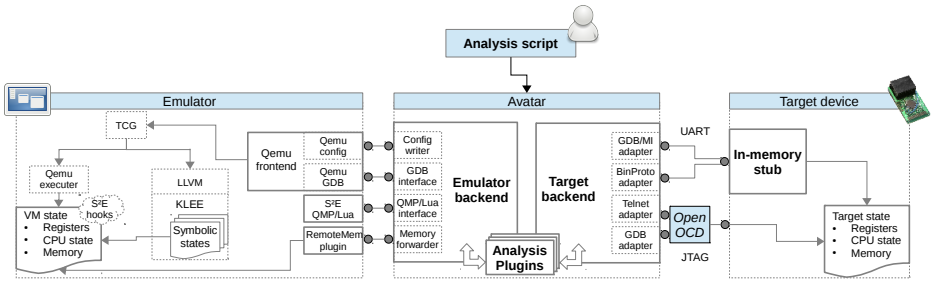


Fig. 2.1: Overview of *Avatar*.

- *Firmware Adaptation*
  Another approach consists in adapting the firmware (or in extracting limited parts of its code) in order to emulate it in a generic emulator. While this is possible in some specific cases, for example with Linux-based embedded devices, this technique does not allow for an holistic analysis and may still be limited by the presence of custom peripherals. Moreover, this approach is not possible for monolithic firmwares that cannot be easily split into independent parts - unfortunately a very common case in low-end embedded systems [88].

In the next section we present our novel hybrid technique based on a combination of the actual hardware with a generic CPU emulator. Our approach allows to perform advanced dynamic analysis of embedded systems, even when very little information is available on their firmware and hardware, or when basic hardware debugging support is not available. This opens the possibility to analyze a large corpus of devices on which dynamic analysis was not possible before.

2.3 Avatar

Avatar[2] is an event-based arbitration framework that orchestrates the communication between an emulator and a target physical device.

Avatar's goal is to enable complex dynamic analysis of embedded firmware in order to assist in a wide range of security-related activities including (but not limited to) reverse engineering, malware analysis, vulnerability discovery, vulnerability assessment, backtrace acquisition and root-cause analysis of known test cases.

2.3.1 System Architecture

The architecture of the system is summarized in Figure 2.1: the firmware code is executed inside a modified emulator, running on a traditional personal computer. Any IO access is then intercepted and forwarded to the physical device, while signals and interrupts are collected on the device and injected into the emulator.

[2]The *Avatar* framework is open-source and available at http://s3.eurecom.fr/tools/avatar.

The internal architecture is completely event-based, allowing user-defined plugins to tap into the data stream and even modify the data as it flows between the emulator and the target.

In the simplest case *Avatar* requires only a backend to talk to the emulator and one to talk to the target system, but more plugins can be added to automate, customize, and enhance the firmware analysis. In our prototype, we developed a single emulator backend. This controls S^{2}E (or Selective Symbolic Execution engine), which is an open-source platform for selective symbolic execution of binary code [72]. It builds on the foundation of Qemu, a very popular open-source system emulator [38]. Qemu supports many processor families such as i386, x86-64, Arm, Mips and many others. Apart from being a processor emulator, Qemu can also mimic the behavior of many hardware devices that are typically attached to the central processor, such as serial ports, network cards, displays, etc.

S^{2}E leverages the intermediate binary code representation of Qemu called Tiny Code Generator (TCG), and dynamically translates from TCG bytecode to Low-Level Virtual Machine (LLVM) bytecode whenever symbolic execution is active [168]. KLEE, the actual symbolic execution engine, is then taking care of exploring the different execution paths and keeps track of the path constraints for each symbolic value [53]. Evaluating possible states exhaustively, for some symbolic input, can be assimilated to model checking and can lead to proving some property about a piece of software [167].

Even though S^{2}E uses the TCG representation of the binary code to generate LLVM code, each processor architecture has its own intricacies that make it necessary to write architecture specific extensions to make S^{2}E work with a new processor architecture. Since our focus was on embedded systems and all the systems we analyzed are ARM systems, we updated and improved an existing incomplete ARM port[3] of S^{2}E, to suit the needs of dynamic analysis of firmware binaries.

To control the execution of code in more detail, S^{2}E provides a powerful plugin interface that allows instrumentation of virtually every aspect of execution. Any emulation event (e.g., translation of a basic block, instruction translation or execution, memory accesses, processor exceptions) can be intercepted by a plugin, which then can modify the execution state according to its needs. This modular architecture let us perform dynamic analysis of firmware behaviour, such as recording and sandboxing memory accesses, performing live migration of subroutines (see Section 2.3.3), symbolically executing specific portion of code as well as detecting vulnerabilities (see Section 2.5).

S^{2}E is connected through three different control interfaces with *Avatar*: the first interface is a GDB debug connection using the GDB serial protocol. *Avatar* is connecting to this interface using a GDB instance controlled via the GDB/MI protocol. This connection is used for fine-grained control over the execution, such as putting breakpoints, single-stepping the execution, and inspecting register val-

[3] Our patches have been submitted to the official S^{2}E project and are currently under review for merging.

ues. The second interface is Qemu's Management Protocol (QMP) interface, a JSON-based request-response protocol. Though detailed virtual machine control is possible through this interface, it is currently only used to dynamically change S^2E's configuration at run time. This is done by accessing S^2E through its Lua interface, which is called from Lua code embedded in the JSON requests. The third interface is a plugin for S^2E that is triggered whenever a memory access is performed. This S^2E plugin then forwards this request to *Avatar*, which in turn handles the memory access (e.g., sends it to *Avatar*'s plugins), or forwards it to the target.

Even though at the moment the only available emulator back-end is for Qemu/S^2E, the emulator interface is generic and allows other emulators to be added easily.

On the target side, we developed three back-ends:

- A back-end that uses the GDB serial protocol to communicate with GDB servers (e.g., a debugger stub installed on the device or a JTAG GDB server).
- A back-end to support low-level access to the OpenOCD's JTAG debugging interface via a telnet-like protocol.
- A back-end that talks to a custom *Avatar* debugger proxy over an optimized binary protocol (which is more efficient than the verbose protocol used by GDB). This proxy can be installed in an embedded device that lacks debugging hardware support (e.g., no hardware breakpoints) or on which such support was permanently deactivated.

The proper target back-end has to be selected by the user based on the characteristics and the debugging functionalities provided by the hardware of the embedded device. For example, in our experiments we used the OpenOCD back-end to connect to the JTAG debugger of the mobile phone and of the Econotag, while we used the *Avatar* proxy to perform dynamic analysis of the hard drive firmware.

To analyze a firmware, an access to the firmware's device is needed. This can be either a debugging link (e.g., JTAG), a way to load software or a code injection vulnerability. In cases where a debugging stub, for example the GDB stub, is used, an additional communication channel, e.g., an UART, is also needed.

2.3.2 Full-Separation Mode

When *Avatar* is first started on a previously unknown firmware, it can be run in what we call *"full-separation mode"*. In this configuration, the entire firmware code is executed in the emulator and the entire (memory) state is kept in the physical device. In other words, for each instruction that is executed by the emulator, the accessed memory addresses are fetched from and written to the real memory of the embedded system. At the same time, interrupts are intercepted by the debugging stub in the physical system and forwarded back to the emulator. Code and memory are perfectly separated, and *Avatar* is responsible to link them together.

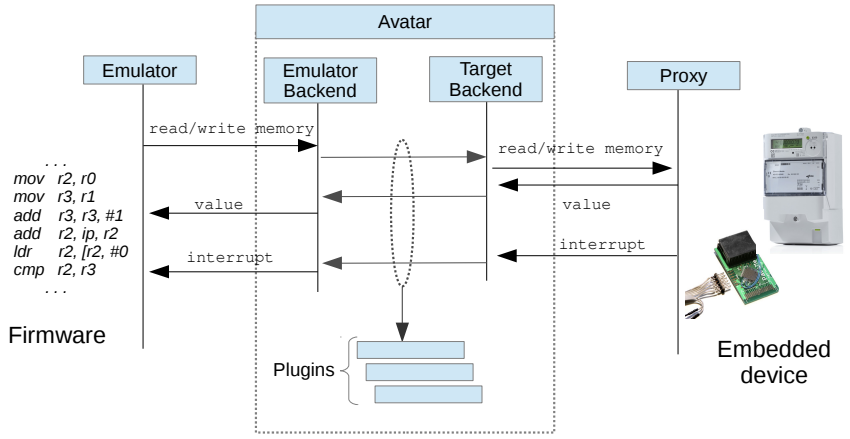


Fig. 2.2: *Avatar* architecture and message exchange in full separation mode.

Even though this technique is in theory capable of performing dynamic analysis on unknown firmwares, it has several practical limitations. First of all, the execution is very slow. Using a serial debug channel at 38400 Baud, the system can perform around five memory accesses per second, reducing the overall emulation speed to the order of tens instructions per second. Even worse, many physical devices have time-critical sections that need to be executed in a short amount of time or the execution would fail, making the system crash. For example, DRAM initialization, timer accuracy and stability checks belong to this category.

Moreover, tight hardware-polling loops (e.g., UART read-with-timeout) become painfully slow in full separation mode. Finally, regular interrupts (e.g., the clock tick) quickly overload the limited bandwidth between the target system and the emulator.

These limitations make the full separation approach viable only to analyze a limited number of instructions or when the user wants to focus only on particular events in more complex firmwares. For this reason, *Avatar* supports arbitrary context-switching between the emulator and the real device.

2.3.3 Context Switching

While it is possible to run the firmware code from beginning to end inside the emulator, sometimes it is more efficient to let the firmware run natively on the target device for a certain amount of time. This allows, for example, to execute the code without any delay until a particular point of interest is reached, skipping through initialization routines that may involve intensive I/O operations or network protocol communications that may need to be performed in real-time. In such cases, it is important to let the target device run the firmware, while still monitoring the execution for regions of code relevant to the current analysis. The ability of *Avatar* to perform arbitrary context switches gives the user the ability to quickly

focus her analysis on a particular section of the code, without the drawbacks of emulating the entire firmware execution.

Starting the analysis at specific points of interest

In this case the firmware starts the execution on the physical device and runs natively until a certain pre-defined event occurs (e.g., a breakpoint is reached or an exception is raised). At this point, the execution on the physical device is frozen and the state (e.g., the content of the CPU registers) is transferred to the emulator, where the execution is resumed. An example of this transition is described in Section 2.6.3, in which the firmware of a mobile phone baseband chip is executed until the phone receives an SMS, and then transferred by *Avatar* in the emulator to perform further analysis.

Returning execution to the hardware

After the required analysis is performed on the emulator, the execution of the firmware can be transferred back to continue on the real device. In this case, any state kept on the virtual environment is copied back to the physical device. Depending on the user's needs, it is possible to switch again to the emulator at a later stage. This approach is used in Section 2.6.1, in which the firmware of a hard disk is started inside the emulator and later transferred back to the disk.

2.3.4 Interrupts Handling

Software interrupts do not present a problem for our framework, since they are issued by the firmware code and the emulator takes care of calling the corresponding interrupt handler directly. However, as shown in Figure 2.2, hardware interrupts need to be trapped in the real hardware and forwarded back to the emulator. In this case, the stub in the embedded system receive the interrupt and forwards them to *Avatar*'s target back-end. Finally, using the emulator back-end, *Avatar* suspends the firmware execution and injects the interrupt in the emulator.

Based on the circumstances in which the interrupt is generated, we distinguish three different cases:

- Hardware interrupts that indicate the completion of a task. These interrupts are issued by a device to indicate that a particular task initiated by the code has been completed. For example, the UART *send* interrupt indicates that the send buffer has been successfully transmitted. This type of interrupts is easy to handle because it just needs to be forwarded from the target to the emulator.

- Periodical hardware interrupts, e.g., the timer notifications. These interrupts can be forwarded to the emulator but their frequency needs to be scaled down to the actual execution speed in the emulator. The equivalent number of instructions between two interrupts should be executed in the emulator as it

would on the target running in native mode. In our current implementation, an *Avatar* plugin detects periodic interrupts and report their information to the user, who can decide how to handle each class. For example, the user can instruct *Avatar* to drop the clock interrupts on the device and just generate them (at the right frequency) on the emulator, thus saving bandwidth and increasing the analysis performance.

- Hardware interrupts that notify of an external event. For example the *receive* interrupt of an UART indicates that new data on the UART buffer is available. The emulation strategy for those interrupts depends on the frequency of the external event. For events that require previous activity (e.g., a request-response protocol where the response triggers an interrupt) a simple forwarding strategy can be used. For unrelated events that happen very frequently (i.e., where the handler in the emulator cannot process the interrupt in time before the next interrupt is generated) the user can choose if she wants to suppress some of them or to handle the interrupt by migrating the handler itself back to the embedded device (see Section 2.4)

While the straightforward interrupt forwarding does not present any problem for *Avatar*, when the user needs to tune the framework to handle specific cases (e.g., regular or very frequent interrupts) the stub needs to be able to distinguish between them. Unfortunately, this task is often difficult.

Interrupts de-multiplexing

In a traditional, x86-based, personal computer there is a standard interrupt controller that handles interrupt lines from each device and peripheral. However, on ARM-based systems there are only two interrupt lines directly attached and visible to the processor: `IRQ` and `FIQ`. Because of this embedded devices often use an interrupt multiplexer (or controller) peripheral that is normally included as an hardware block ("IP core") on the same chip. The disadvantage for a user is that at the point where the interrupt vector routine is called, all interrupt signals are still multiplexed together. The driver for a particular interrupt multiplexer will then query the underlying hardware multiplexer to identify which line was actually triggered and then forward the event to the handler registered for this interrupt.

Now, suppose the user wants to instruct *Avatar* to suppress a particular interrupt on the device (e.g., the timer), while still letting through the ones associated to important hardware events that need to be forwarded to the emulator. In this case, the proxy needs to take a decision based on the interrupt type which is unfortunately not available when the interrupt is received.

In this case, the user needs to disassemble the interrupt vector handler, and follow the code flow until the code of the interrupt controller driver branches into different functions that handle each device's interrupt. At this point, she can specify these program points to *Avatar* that can terminate the interrupt vector's execution and signal to the proxy that an interrupt has been identified. The proxy then sends

the interrupt event to *Avatar*. Now the target backend of *Avatar* can suppress a particular interrupt by instructing the proxy to drop the corresponding event.

2.3.5 Replaying Hardware Interaction

It is quite common for a firmware to have several sections that require only a limited interaction with dedicated peripherals. In this case, the I/O operations can be recorded by *Avatar* and transparently replayed during the next execution of the firmware.

This allows the user to test the firmware without the bottleneck of the interaction with the physical device. In this mode of operation the firmware itself or parts of it (e.g., applications) can be significantly changed, as long as the order of I/O interactions is not modified. This is a major advantage over resuming a snapshot, which requires the full code path until the snapshot point to be executed to ensure that peripherals are in the state the snapshot expects them to be in.

2.4 Overcoming the limits of Full Separation

The techniques introduced in the previous section are enough to perform dynamic analysis on small portions of a firmware code. However, sometimes the internals and behavior of the system are completely unknown. In those cases, it can be very useful to perform the analysis on larger portions of the binary, or, in the extreme case, on the entire firmware.

In this case, the performance of *Avatar* running in full separation mode poses a great limitation to the usability of our framework. To overcome this problem, in this section we present two techniques designed to overcome the limits of full separation by moving part of the code to the physical device and part of the memory to the emulator. This results in a considerable reduction in the number of messages forwarded by *Avatar* between the emulator and the target, and therefore a large improvement in the overall performance of the analysis system.

2.4.1 Memory Optimization

Forwarding all memory accesses from the emulator to the target over a limited-bandwidth channel like UART or JTAG incurs in a heavy performance penalty. For example, in our experiments an average of five instructions per second were executed using the GDB stub through a 38400 baud UART connection.

The reason why memory operations need to be forwarded in the first place is that different embedded systems typically have different mappings of addresses to memory regions. Some of these memory regions are used for code (in RAM, ROM or Flash memories), stack and heap, but one or several regions will be used to access registers of physical peripherals through Memory-Mapped I/O (MMIO). In this case, any I/O operation on those areas is equivalent to sending and receiving data from an external device. If these address ranges are known, the user can configure *Avatar* to keep every read-only memory (such as the code segment) on the

Access type	Read	Write	Cumulative
Code	61,632	-	61,632
Stack & data	646	1,795	64,073
I/O	3,614	2,097	69,784

Table 2.1: Number of memory accesses grouped by memory regions for the HDD bootloader.

emulator. Read-write memory regions can also be marked as local to the emulator, but modifications to them need to be tracked by *Avatar* to be able to transfer those changes to the target at a later context switch. In fact, when an emulator-to-target context switch happens, all modified local memory ("dirty memory") needs to be copied to the target before the execution can resume on the embedded device.

However, in most of the cases the user does not know a priori which area of memory is assigned to I/O. For this reason, *Avatar* includes an automated memory optimization plugin that monitors the execution in the emulator and automatically identifies the regions that do not require access to the hardware. This includes the stack (easily identified by the execution of stack-related operations) and the code segment (identified by the values of the program counter). For any other area, *Avatar* starts by forwarding the read and write operations to the target device. It then keeps track of the values that are returned and applies a simple heuristic: if the target always returns the value that was previously written by the firmware code (or if it always returns the same value and it is never written by the firmware) then it is probably not assigned to a memory mapped device.

Table 2.1 shows an example of how many memory accesses could be saved by keeping memory regions local to the emulator: transferring the code region to the emulator would save 61,632 memory accesses (88%). Moving the stack and data region in local memory as well would save 64,073 memory accesses (92%). Only the I/O accesses cannot be moved to the emulator's memory.

2.4.2 Selective Code Migration

So far, we assumed that the firmware is either running entirely inside the emulator, or entirely on the embedded device. The user can instruct *Avatar* to switch from one mode to the other when certain conditions are met, but such context switches are time consuming.

In this section we present a fine-grained solution that allows the user to migrate only parts of the firmware code back to the target. This technique allows to overcome two limitations of the full-separation mode. Some code blocks need to be executed atomically, for example when there are timing constraints on the code. We will describe such a case in Section 2.6.1, where we encountered a function that read the timer twice and waited for the difference to be below a certain limit. Another example is when delays introduced by *Avatar* would lead the target in an

invalid state. We encountered such a case during the DRAM initialization of the HDD, as shown in Section 2.6.1).

The second limitation addressed by selective code migration is related to the analysis performance. In fact, certain functions (e.g., polling loops and interrupt handlers) can be executed significantly faster when run natively on the target.

In the current *Avatar* prototype, code migration is supported at a function level. In this case, the code can be copied to its location in the target's memory without modification. Its exit points are then replaced by breakpoints, and the virtual machine register state is transferred from the emulator to the target. The execution is resumed on the target until one of the exit breakpoints is triggered, and at that point the state is transferred back to the emulator. This transition is much faster than a complete context switch, since *Avatar* only needs to transfer few bytes and not the entire content of the memory.

Even though this simple technique is enough to circumvent critical code regions in several real world scenarios, it neglects some difficulties that may affect code migration. First, the code may read or write arbitrary memory locations associated, for example, with global variables. *Avatar* keeps track of those locations, copy their content over to the target before the execution, and copy written locations back after the execution. Second, the code may use instructions that change the control flow in unforeseen ways, like software interrupts, processor mode changes, and indirect jumps.

Our framework prototype addresses these issues by performing an on-the-fly static analysis. When a function is selected for code migration, *Avatar* disassembles its code using the `llvm-mc` disassembler. The result is then analyzed to identify critical instructions. In this way, we can predict memory accesses outside the function stack, compute the control flow of the code and verify that no instructions can escape from this computed control flow. As we describe in Section 2.6, this technique is sufficient to migrate small, atomic functions. However, we plan to extend the capabilities of the code migration system to apply transformations to the code. On the one hand, those transformations will allow to ensure that instructions which are not statically verifiable (e.g., indirect jumps) will not escape the proxy's sandbox. On the other hand, it can be used to track memory accesses, so that only the modified ("dirty") part of the state needs to be copied back from the target to the emulator when a context switch happens. Those critical instructions will be replaced with instrumentation code that calls functions in proxy, which will handle them in a safe way.

2.5 Extending Avatar

Avatar's architecture is designed to be modular and its base framework can be easily customized to fit different analysis scenarios. We chose S^2E as default *Avatar* emulator back-end because it offers many hooks and manipulation facili-

ties on top of QEMU which facilitates the development of custom dynamic analysis plugins.

In this section, we show an example of an *Avatar* extension: we built upon its core capabilities to support selective symbolic execution. For this we add several features and plugins to the ARM port of S^2E. Moreover, we believe the symbolic execution engine provides a super-set of the capabilities needed to implement taint analysis, even though a targeted plugin could be needed to perform concrete data tracking and taint analysis in a more lightweight way.

In the rest of this section we describe the technique *Avatar* employs to fully exploit the symbolic engine of S^2E and perform selective symbolic execution on unmodified portions of firmware blobs. Moreover, we show how we use our extended version of S^2E in *Avatar* to dynamically detect potential control flow corruption vulnerabilities by injecting and tracking symbolic inputs.

2.5.1 Injecting Symbolic Values Into the Firmware's Execution Flow

In the field of program testing, symbolic execution is a technique employed to improve code coverage by using symbols as input data (instead of concrete values) and keeping track of constraints upon their manipulation or comparison (c.f. [218]). The result of symbolic evaluation is an execution tree, where each path is a possible execution state that can be reached by satisfying the constraints associated to each symbolic value.

S^2E further develops this concept by performing selective symbolic execution, i.e., by restricting the area of symbolic execution to specific code portions and treating only specific input data as symbolic [72]. This greatly helps to speedup the analysis process (as symbolic execution of code results in significant slowdowns) and to drive the exhaustive symbolic exploration into selected regions of code. This process requires *Avatar* to control the introduction of symbolic values into S^2E, in place of existing real values.

The remote memory interface between S^2E and *Avatar*, as introduced in Section 2.3, ensures that only concrete values reach the real hardware through *Avatar*. Symbolic values remain therefore confined to the emulation domain. If a symbolic value is about to be written to the target hardware, the remote memory interface in S^2E performs a forced concretization before forwarding it. Such symbolic value concretizations happen in two stages. First, all the constraints associated with the value are retrieved and evaluated by the integrated SAT-solver. Second, a single example value which satisfies all the constraints is forwarded to *Avatar* to be written on the target.

On the one hand, making *Avatar* handle only concrete values leaves it as a controller with a simpler external view of S^2E and avoids having to keep track of execution paths and paths conditions twice. On the other hand, this choice brings the minor drawback that *Avatar* has no direct control on symbolic execution, which is instead under the control of S^2E/KLEE.

We designed a simple plugin for detecting arbitrary execution conditions. It relies on the following heuristics as signs of possibly exploitable conditions:

- a symbolic address being used as the target of a load or store instruction,
- a symbolic address being leaked into the program counter (e.g., as the target of a branch),
- a symbolic address being moved into the stack pointer register.

In order to selectively mark some input data as symbolic, two different approaches can be taken: either modify the binary code (or the source code, if available) to inject custom instructions into the firmware, or dynamically instrument the emulation environment to specify the scope of symbolic analysis at run-time. The first approach requires some high-level knowledge of the firmware under analysis (e.g., access to source code) and the guarantee that injecting custom instructions into firmware code would not affect its behavior. Examples include the Android Dalvik VM, whose source code can be modified and rebuilt to enable transparent analysis of pristine Java bytecode with S^2E [160].

Since we did not want to limit *Avatar* to this scenario, we decided to follow the second approach, which requires to extend the symbolic engine and the *Avatar* framework. Such extensions should know when symbolic execution has to be triggered and where symbolic values should be injected.

This choice leads to two major advantages:

- *Firmware Integrity*
  The binary code is emulated as-is, without injecting custom opcodes or performing recompilation. This guarantees that the emulated code adheres to the original firmware behavior (i.e., no side-effects or bugs are introduced by the intermediate toolchain)
- *Programmatic Annotation*
  The control and data flow of firmware emulation can be manipulated and annotated with symbolic meta-data in an imperative way. A high-level language (Lua) is used to dynamically script and interact with current emulation environment, as well as introducing and tracing symbolic meta-data.

For this we first completed the port of S^2E to the ARM architecture in order to have complete symbolic execution capabilities, then we ported the Annotation plugin to the ARM architecture. The Annotation plugin lets the user specify a trigger event (e.g., a call/return to a specific subroutine or the execution of code at a specific address), and a Lua function to be executed upon the event. A simple API is then provided to allow for manipulation of the S^2E emulation environment directly from the Lua code. *Avatar* provides direct channels to dynamically control the emulation flow via QMP command messages. These channels can also be used to inject Lua code at run-time, in order to dynamically generate annotations which

depend on the current emulation flow and inject them back into S^2E. Once symbolic values are introduced in the execution flow, S^2E tracks them and propagates the constraints.

Symbolic analysis via Lua annotations is intended to be used as a tool for late stage analysis, typically to ease the discovery of flaws in logic-handling code, with hand-made Lua analysis code directly provided by the user. It can be employed in both full separation mode and context switching, as soon as code execution can be safely moved to the emulator (e.g., outside of raw I/O setup routines, sensors polling). This normally happens after an initial analysis has been done with *Avatar* to detect interesting code and memory mappings.

A similar non-intrusive approach has already been used in a x86-specific context, to test and reverse-engineer the Windows driver of a network card [70]. To the best of our knowledge, however, this technique has never been applied before to embedded devices. In the context of firmware security testing, annotations can be used in a broad range of scenarios. In Section 2.6, we present how we applied this technique to different technologies and devices, to perform dynamic analysis of widespread embedded systems such as hard drives, GSM phones, and wireless sensors.

2.5.2 Symbolically Detecting Arbitrary Execution Conditions

When dealing with modern operating systems, an incorrect behavior in a user-space program is often detected because an invalid operation is performed by the program itself. Such operations can be, for example, an unauthorized access to a memory page, or the access to a page that is not mapped in memory. In those cases, the kernel would catch the wrong behavior and terminate the program, optionally triggering some analysis tools to register the event and collect further information that can later be used to identify and debug the problem. Moreover, thanks to the wide range of exploit mitigation techniques in place today (DEP, canaries, sandboxing and more), the system is often able to detect the most common invalid operations performed by userspace processes.

When dealing with embedded systems, however, detecting misbehavior in firmware code can be more difficult. The observable symptoms are not always directly pinpointed to some specific portion of code. For example, many firmware are designed for devices without a Memory Management Unit (MMU) or Memory Protection Unit (MPU) or are just not using them. In such a context, incorrect memory accesses often result in subtle data corruption which sometimes leads to erratic behaviors and rare software faults, such as random events triggering, UI glitches, system lock or slowdown [82]. For this reason, it is common for embedded devices to have a hardware watchdog in charge of resetting the device execution in case of any erratic behavior, e.g., a missed reply to timed watchdog probes.

For these reasons, detecting incorrect execution inside the emulation is easier when some OS support can be used for co-operation (e.g., a *Blue Screen Of Death* interceptor for Windows kernel bugs is implemented in S^2E). On the other hand,

catching such conditions during the emulation of an embedded device firmware is bound to many system-specific constraints, and require additional knowledge about the internal details of the firmware under analysis.

However, *Avatar* does not rely on the knowledge of any specific operating system or the fact that a MMU is used. Instead, it aims at detecting a larger range of potentially critical situations which may result in control flow hijacking of firmware code, by using a technique similar to the one employed by AEG [26].

All three conditions may lead to false positives, when the variable is symbolic but strongly constrained. Therefore, once such a condition is detected the constraints imposed on the symbolic variables must be analyzed: the less constrained is the result, the higher is the chance of control flow corruption. Intuitively, if the constraints are very loose (e.g., a symbolic program counter without an upper bound) then the attacker may obtain enough control on the code to easily exploit the behavior. In addition to this, tight constraints are sometimes encountered in legitimate cases (e.g., access to an array with a symbolic but constrained index such as with a jump table), and are not relevant for the purpose of security analysis.

When an interesting execution path is detected by the above heuristic, the state associated to the faulty operation is recorded and the emulation is terminated. At this point a test-case with an example input to reach this state is generated, and the constraints associated with each symbolic value are stored to be checked for false positives (i.e., values too strictly bound).

Automatically telling normal constraints apart from those that are a sign of a vulnerability is a complex task. In fact it would require knowledge of the program semantics that were lost during compilation (e.g., array boundaries). Such knowledge could be extracted from the source code if it is available, or might be extrapolated from binary artifacts in the executable itself or the build environment. In such cases, specific constraints could be fed into *Avatar* by writing appropriate plugins to parse them, for example by scanning debug symbols in a non-stripped firmware (e.g., a DWARF parser for ELF firmwares) or by reading other similar symbols information.

Finally, *Avatar* could highly benefit from a tighter coupling with a dynamic data excavator, helping to reverse engineer firmware data structures [81]. In particular, the heuristic proposed in Howard [227] for recovering data structures by observing access patterns under several execution cycles could be easily imported into the *Avatar* framework. Both tools perform binary instrumentation on top of QEMU dynamic translation and make use of a symbolic engine to expand the analyzed code coverage area.

2.5.3 Limitations of state synchronization

Our current implementation of the synchronization between device state and emulator state works well in general, but is difficult in some special cases.

First it is difficult to handle DMA memory accesses in our current model. For example, the firmware can send a memory address to a peripheral and request data

to be written there. The peripheral will then notify the firmware of the request's completion using an interrupt. Since *Avatar* does not know about this protocol between firmware and peripheral, it will not know which memory regions have been changed. On newer ARM architectures with caches, *data synchronization barrier* or *cache invalidation* instructions might be taken as hint that some memory region has been changed by DMA.

Second, if code is executed on the device, *Avatar* is currently incapable of detecting which regions have been modified. In consequence, whenever memory accesses of the code run on the device are not predictable by static analysis, we need to transfer the whole memory of the device back to the emulator on a device-to-emulator state switch. We plan to address this issue by using checksumming to detect memory region changes and minimize transferred data by identifying smallest changed regions through binary search.

Third, when *Avatar* performs symbolic execution, symbolic values are confined to the emulator. In case that a symbolic value needs to be concretized and sent to the device, a strategy is needed to keep track of the different states and I/O interactions that were required to put the device in that state. This can be performed reliably by restarting the device and replaying I/O accesses. While this solution ensures full consistency, it is rather slow.

2.6 Evaluation

In this section we present three case studies to demonstrate the capabilities of the *Avatar* framework on three different real world embedded systems. These three examples by no means cover all the possible scenarios in which *Avatar* can be applied. Our goal was to realize a flexible framework that a user can use to perform a wide range of dynamic analysis on known and unknown firmware images.

As many other security tools (such as a disassembler or an emulator), *Avatar* requires to be configured and tuned for each situation. In this section, we try to emphasize this process, in order to show all the steps a user would follow to successfully perform the analysis and reach her goal. In particular, we will discuss how different *Avatar* configurations and optimization techniques affected the performance of the analysis and the success of the emulation.

Not all the devices we tested were equipped with a debug interface, and the amount of available documentation varied considerably between them. In each case, human intervention was required to determine appropriate points where to hook execution and portions of code to be analyzed, incrementally building the knowledge-base on each firmware in an iterative way. A summary of the main characteristics of each device and of the goal of our analysis is shown in Table 2.2.

2.6.1 Analysis of the Mask ROM Bootloader of a Hard Disk Drive

Our first case study is the analysis of a masked ROM bootloader and the first part of the secondary bootloader of a hard disk drive.

	Target device	Manufacturer	System-on-Chip	CPU	Debug access	Analyzed code	Scope of analysis
Exp 2.6.1	Hard disk	*undisclosed*	unknown	ARM966	Serial port	Bootloader	Backdoor detection
Exp 2.6.2	ZigBee sensor	Redwire Econotag	MC13224	ARM7TDMI	JTAG	ZigBee stack	Vulnerability discovery
Exp 2.6.3	GSM phone	Motorola C118	TI Calypso	ARM7TDMI	JTAG	SMS decoding	Reverse engineering

Table 2.2: Comparison of experiments described in Section 2.6.

The hard disk we used in our experiment is a commercial-off-the-shelf SATA drive from a major hard disk manufacturer. It contains an ARM 966 processor (that implements the ARMv5 instruction set), an on-chip ROM memory which contains the masked ROM bootloader and some library functions, an external serial flash that is connected over the SPI bus to the processor, a dynamic memory (SDRAM) controller, a serial port accessible through the master/slave jumpers, and some other custom hardware that is necessary for the drive's operation. The drive is equipped with a JTAG connection, but unfortunately the debugging features were disabled in our device. The hard drive's memory layout is summarized in Figure 2.4.

The stage-0 bootloader executed from mask ROM is normally used to load the next bootloader stage from a SPI-attached flash memory. However, a debug mode is known to be reachable over the serial port, with a handful of commands available for flashing purposes. Our first goal was to inject the *Avatar* stub through this channel to take over the booting process, and later use our framework for deeper analysis of possible hidden features (e.g., backdoors reachable via the UART).

The first experiment we performed consisted of loading the *Avatar* stub on the drive controller and run the bootloader's firmware in full separation mode. This mimics what a user with no previous knowledge of the system would do in the beginning. In full separation mode, all memory accesses were forwarded through the *Avatar* binary protocol over the serial port connection to the stub and executed on the hard drive, while the code was interpreted by S^2E. Because of the limited capacity of the serial connection, and the very intensive I/O performed at the beginning of the loader (to read the next stage from the flash chip), only few instructions per second were emulated by the system. After 24 hours of execution without even reaching the first bootloader menu, we aborted the experiment.

In the second experiment we kept the same setting, but we used the memory optimization plugin to automatically detect the code and the stack memory regions and mark them as local to the emulator. This change was enough to reach the bootloader menu after approximately eight hours of emulation. Though considerably faster than in the first experiment, the overhead was still unacceptable for this kind of analysis.

Since the bottleneck of the process was the multiple read operations performed by the firmware to load the second stage, we configured *Avatar* to replay the hardware interaction from disk, without forwarding the request to the real hardware.

Fig. 2.3: The disk drive used for experiments. The disk is connected to a SATA (Data+Power) to USB interface (black box on the right) and its serial port is connected to a TTL-serial to USB converter (not shown) via the 3 wires that can be seen on the right.

In particular, we used the trace of the communication with the flash memory from the second experiment to extract the content of the flash memory, and dump it into a file. Once the read operations were performed locally in the emulator, the bootloader menu was reached in less than four minutes.

At this point, we reached an acceptable working configuration. In the next experiment, we show how *Avatar* can be used in conjunction with the symbolic execution of S^2E to automatically analyze the communication protocol of the hard drive's bootloader and detect any hidden backdoor in it.

We configured *Avatar* to execute the hard drive's bootloader until the menu was loaded, and then replace all data read from the serial port register by symbolic values. As a result, S^2E started exploring all possible code paths related to the user input. This way, we were able to discover all possible input commands, either legitimate or hidden (which may be considered backdoors), that could be used to execute arbitrary code by using S^2E to track when symbolic values were used as address and value of a memory write, and when the program counter would become symbolic. With similar methodologies, a user could use symbolic execution to

DS	Use a minimal version of the Motorola S-Record binary data format to transmit data to the device
AP <addr>	Set the value of the address pointer from the parameter passed as hexadecimal number. The address pointer provides the address for the read, write and execute commands.
WT <data>	Write a byte value at the address pointer. The address pointer is incremented by this operation. The reply of this command depends on the current terminal echo state.
RD	Read a byte from the memory pointed to by the address pointer. The address pointer is incremented by this operation. The reply of this command depends on the current terminal echo state.
GO	Execute the code pointed to by the address pointer. The code is called as a function with no parameters, to execute Thumb code one needs to specify the code's address + 1.
TE	Switch the terminal echo state. The terminal echo state controls the verbosity of the read and write commands.
BR <divisor>	Set the serial port baud rate. The parameter is the value that will be written in the baud rate register, for example "A2" will set a baudrate of 38400.
BT	Resume execution with the firmware loaded from flash.
WW	Erase a word (4 bytes) at the address pointer and increment address pointer.
?	Print the help menu showing these commands.

Table 2.3: Mask ROM bootloader commands of the hard drive. In the left column you can see the output of the help menu that is printed by the bootloader. In the right column a description obtained by reverse engineering with symbolic execution is given.

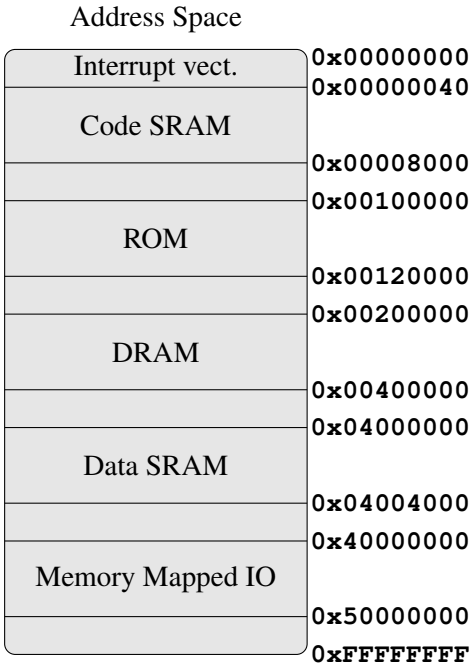


Fig. 2.4: Hard drive memory layout.

automatically discover backdoors or undocumented commands in input parsers and communication protocols.

In order to conduct a larger verification of the firmware input handler, we were also able to recover all the accepted commands and verify their semantics. Since the menu offered a simple online `help` to list all the available commands, we could demonstrate that *Avatar* was indeed able to automatically detect each and all of them (the complete list is reported in Table 2.3). In this particular device, we verified that no hidden commands are interpreted by the firmware and that a subset of the commands can be used to make arbitrary memory modifications or execute code on the controller, as documented.

However, we found that the actual protocol (as extracted by symbolic analysis) is much looser than what is specified in the `help` menu. For example the argument of the 'AP' command can be separated by any character from the command, not only spaces. It is also possible to enter arbitrarily long numbers as arguments, where only the last 8 digits are actually taken into account by the firmware code.

After the analysis of the first stage was completed, we tried to move to the emulation of the second stage bootloader. At one point, in what turned out to be the initialization of the DRAM, the execution got stuck: the proxy on the hard drive would not respond any more, and the whole device seemed to have crashed. Our guess was that the initialization writes the DRAM timings and needs to be performed atomically. Since we already knew the exact line of the crash from the execution trace, it was easy to locate the responsible code, isolate the corresponding

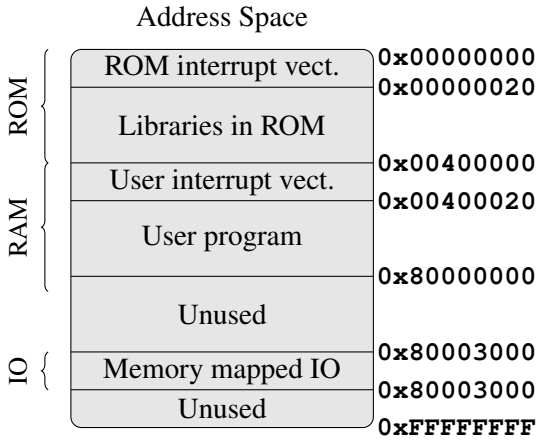


Fig. 2.5: Econotag memory layout (respective scales not respected).

function, and instruct *Avatar* to push its code back to be executed natively on the hard drive.

In a similar manner, we had to mark few other functions to be migrated to the real hardware. One example is the timer routine, which was reading the timer value twice and then checked that the difference was below a certain threshold (most probably to ensure that the timer read had not been subject to jitter). Using this technique, in few iterations we managed to arrive at the final *Avatar* configuration that allowed us to emulate the first and second stages up to the point in which the disk would start loading the actual operating system from the disk's platters.

2.6.2 Finding Vulnerabilities Into a Commercial Zigbee Device

The Econotag, shown in Figure 2.6, is an all-in-one device for experimenting with low power wireless protocols based on the IEEE 802.15.4 standard [142], such as Zigbee or 6lowpan [193]. It is built around the MC13224v System on a Chip from Freescale. The MC13224v [211] is built upon an ARM7TDMI microcontroller, includes several memories, peripherals and has an integrated IEEE 802.15.4 compatible radio transceiver. As it can be seen in Figure 2.5, the device includes 96KB of RAM memory, 80 KB of ROM and a serial Flash for storing data. The ROM memory contains drivers for several peripherals as well as one to control the radio, known as *MACA* (MAC Accelerator), which allows to use the dedicated hardware logic supporting radio communications (e.g., automated ACK and CRC computation).

The goal of this experiment is to detect vulnerabilities in the code that process incoming packets. For this purpose, we use two Econotag devices and a program from the Freescale demonstration kit that simulates a wireless serial connection (wireless UART [115]) using the *Simple MAC* (SMAC [116]) proprietary MAC layer network stack. The program is essentially receiving characters from its UART and transmitting them as radio packets as well as forwarding the charac-

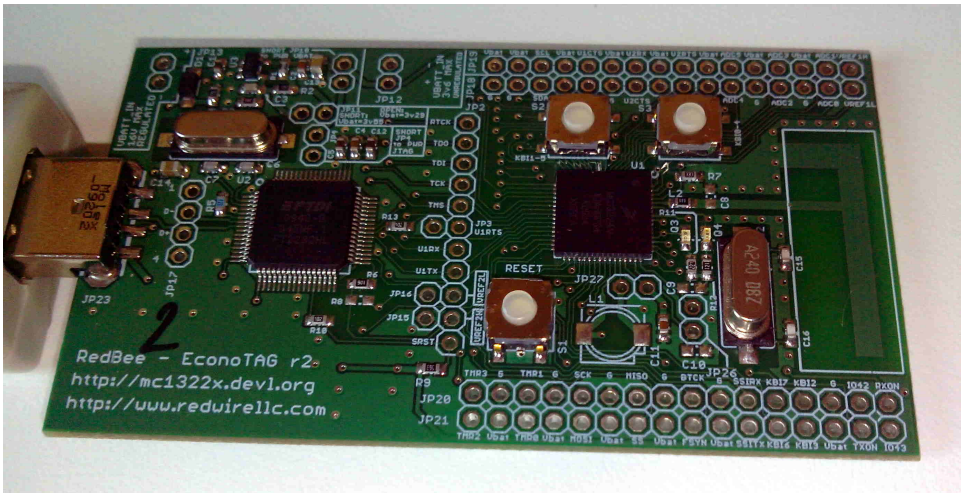


Fig. 2.6: The Econotag device. From left to right: the USB connector, serial and JTAG to USB converter (FTDI), Freescale MC13224v controller and the PCB 2.4 GHz antenna.

ters received on the radio side to its serial port. Two such devices communicating together essentially simulate a wireless serial connection.

The data received from the radio is buffered before being sent to the serial port. For demonstration purposes, we artificially modified this buffer management to insert a vulnerability: a simple stack-based buffer overflow. We then compiled this program for the Econotag and installed it on both devices.

Avatar was configured to let the firmware run natively until the communication between the two devices started. At this point, *Avatar* was instructed to perform a context switch to move the run-time state (registers and data memory) of one of the devices to the emulator. At this point, the execution proceeded in full separation mode inside the emulator using the code loaded in ROM memory (extracted from a previous dump), and the code loaded in RAM memory (taken from the application). Every I/O access was forwarded to the physical device through the JTAG connection.

The emulator was also configured to perform symbolic execution. For this purpose, we used *annotations* to mark the buffer that contains the received packet data as symbolic. Then, we employed a state selection strategy to choose symbolic states which maximize the code coverage, leading to a thorough analysis of the function.

On the first instruction that uses symbolic values in the buffer, S^2E would switch from concrete to symbolic execution mode. Execution will fork states when, for example, conditional branches that depend on such symbolic values are evaluated. After exploring 564 states, and within less than a minute of symbolic execution, our simple *arbitrary execution detection module* detected that an uncon-

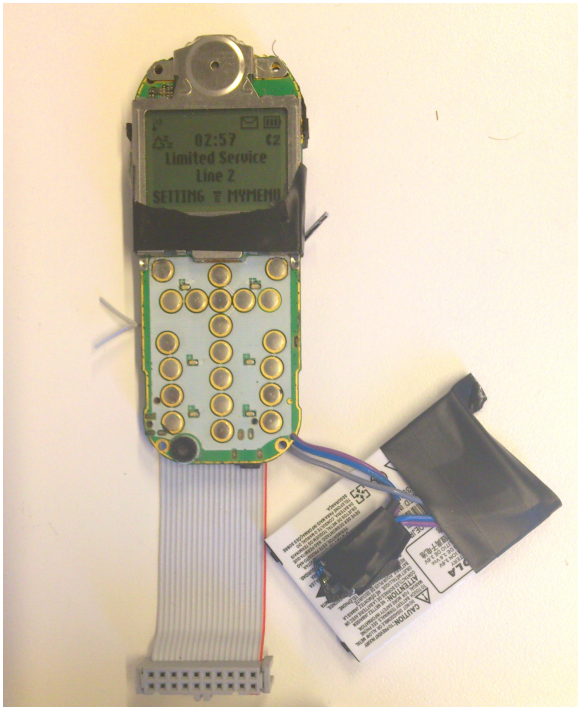


Fig. 2.7: The Motorola C118. The clip-on battery (on the right) has been wired to the corresponding power pins, while the ribbon cable is connected to the JTAG pads reachable on the back (not shown).

strained symbolic value was used as a return address. This confirmed the detection of the vulnerability and also provided an example of payload that triggers the vulnerability.

We also used *Avatar* to exhaustively explore all possible states of this function on a program without the injected vulnerability, and confirmed the absence of control flow corruption vulnerabilities that could be triggered by a network packet (that our simple arbitrary execution detection module could detect).

2.6.3 Manipulating the GSM Network Stack of a Common Feature Phone

Our final test-case is centered on the analysis of the firmware of a common GSM feature phone. In contrast with most recent and advanced mobile phones and smartphones, feature phones are characterized by having one single embedded processor for both the network stack (i.e., GSM baseband capabilities) and the Human-to-Machine Interface (HMI: comprising the main Graphical User Interface, advanced phone services, and miscellaneous applications). As such, there is no clear code separation between different firmware sections. On these phones, typically a real-time kernel takes care of scheduling all the tasks for the processes

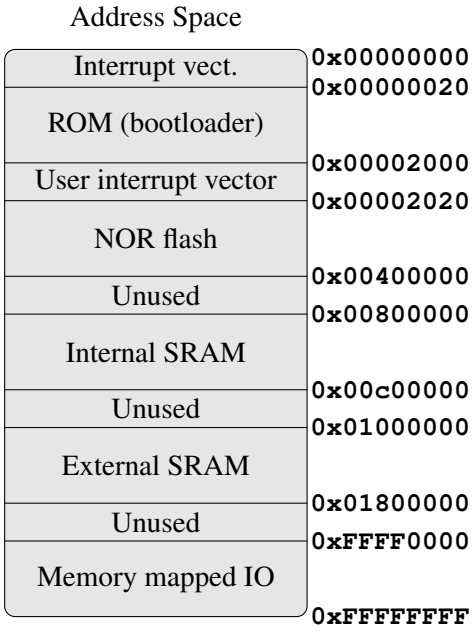


Fig. 2.8: Motorola C118 memory layout (respective scales not respected).

currently in execution. These are executed in the same context and have shared access to the whole physical memory as well as memory-mapped I/O.

GSM baseband stacks have already been shown to have a large potentially exploitable attack surface [252]. Those stacks are developed by few companies worldwide and have many legacy parts which were not written with security in mind, and in particular were not considering attacks coming from the GSM infrastructure [253].

For our experiment, we used a Motorola C118, which is a re-branded version of the Compal E88 board also found in other Motorola feature phones. This board makes use of the *Texas Instruments "Calypso"* digital baseband, which is composed of a mask-ROM, a DSP for GSM signal decoding, and a single ARM7TDMI processor. It also includes several peripherals such as an RTC clock, a PWM generator for controlling the lights and buzzer as well as a memory mapped UART as shown in Figure 2.8. Some board models have JTAG and UART ports available, which are from time to time left enabled by manufacturers to simplify servicing devices. In our case, we gained access to the JTAG port and used an adapter to bridge communication between *Avatar* and the hardware, as shown in Figure 2.7.

Some specification documents on the *Calypso* chipset have been leaked in the past, leading to the creation of home-brew phone OS that could be run on such boards. As part of the Osmocom-BB project, most of the platform has been reversed and documented, and it is now possible to run a free open-source software GSM stack on it [8]. However, we conducted our experiments on the original Motorola firmware, in order to assess the baseband code of an unmodified phone.

Moreover, as the GSM network code is provided as a library by the baseband manufacturer, there is an higher chance that flaws affecting the library code would also be present in a broader range of phones using baseband chips from that same vendor.

The phone has a first-stage bootloader executed on hardware reset, which can be used to re-flash the firmware. After phone setup, execution continues to the main firmware, which is mainly composed of the Nucleus RTOS, the TI network stack library, and of third-party code to manage the user interface. The phone bootloader can be analyzed using *Avatar* in a similar way as the one already described for the hard disk in Section 2.6.1 to discover flashing commands, hidden menus and possible backdoors. However, the bootloader revealed itself to be simpler than the hard drive one, supporting only a UART command to trigger firmware flashing and executing the flashed firmware, or continuing execution after a timeout expiration.

For this reason, we focused on the analysis of the GSM network stack, and in particular on the routines dedicated to SMS decoding. It has already been shown in the past how maliciously crafted SMS can cause misbehavior, ranging from UI issues to phone crashes [195]. However, due to the lack of a dynamic analysis platform to analyze embedded devices, previous studies relied on blind SMS fuzzing. Our experiment aims at improving the effectiveness of SMS fuzzing to detect remotely exploitable execution paths.

In this scenario, *Avatar* was configured to start the execution of the firmware on the real device, and switch to the emulator once the code reached the SMS receiving state (e.g., by sending a legitimate SMS to it through the GSM network). *Avatar* was then used to selectively emulate and symbolically explore the decoding routines. As a result of this exploration, a user is able to detect faulty conditions, to determine code coverage due to different inputs and to recover precise input constraints to drive the firmware execution into interesting areas.

In this context, *Avatar* uses the JTAG connection to stop the execution on the target and later perform all synchronization steps between the emulator and the target. All memory and I/O accesses through JTAG are traced by *Avatar* to let the user identify address mappings. When the phone reaches the SMS receiving state, a target-to-emulator context switch happens and the phone's state is transferred into S^2E. Using address mapping information previously recovered through *Avatar*, just the relevant memory is moved into S^2E (e.g., portions of code and the execution stack), while remaining memory is kept on the target and forwarded on-the-fly by *Avatar* (e.g., I/O regions). On this device, no selective code migration was required.

Using this *Avatar* configuration, the SMS payload can be intercepted in memory and marked as symbolic by employing the techniques shown in Section 2.5. In particular, we wrote Annotation functions to be triggered before entering the decoding routines and we then proceeded to selectively mark some bytes of their input arguments as symbolic. The S^2E plug-in for Arbitrary Execution Detection has been employed to isolate interesting vulnerable cases, while other execution paths were killed upon reaching the end of the decoding function.

The symbolic execution experiments have been performed over several days, with the ones with larger number of symbolic inputs taking up to 10 hours before filling up 60 GB of available memory. In such case, we observed more than 120,000 states being spawned according to different constraints solving. Unfortunately, and contrary to the other experiments, the GSM network stack proved to be way too complex to be symbolically analyzed without prior knowledge on the high-level structure of the code. The analysis was clobbered by an explosion of possible states due to many forks happening in pointer-manipulating loops. *Avatar* was able to symbolically explore 42 subroutines executed during SMS decoding, without detecting any exploitable conditions. However, it was able to highlight several situations of user-controlled memory load, which were unfortunately too strictly constrained to be exploited, as discussed in Section 2.5.2.

State explosion is a well-known limitation of symbolic execution. To mitigate the problem, a user may need to define heuristics to avoid an excessive resource consumption. This could be done, for example, by employing more aggressive state selectors to enhance code coverage, and actively prune states by looking at loops invariants [216]. However, this optimizations are outside the scope of our paper. The objective of our experiments are, in fact, limited to prove that *Avatar* can be used to perform dynamic analysis of complex firmware of embedded devices.

2.7 Related work

The importance of porting dynamic analysis techniques to different platforms has been discussed by Li and Wang [172], who proposed a set of tools built on top of IDA Pro and the REIL Intermediate Language to perform symbolic execution in a portable way.

However, embedded systems have long been recognized to be a difficult target for debugging and dynamic analysis. SymDrive [212] presents a technique based on symbolic execution to test Linux and FreeBSD device drivers without their device present. However, by replacing every input with a symbolic value, this approach is hard to scale and would suffer of state explosion on any real world firmware. In [70], Chipounov and Candea present REVNIC, a tool based on S^2E [72] that helps to reverse engineer network device drivers. As a case study the authors port a Windows device driver for a common network card to a different Operating System. While the presented approach is interesting, it relies on the presence (and extension) of the emulated device and PCI bus in QEMU. Instead, *Avatar* is hardware agnostic, as it does not need to know how peripherals are connected, mapped and accessed. Instead I/O can be simply forwarded to the real target and I/O related code directly executed there.

Cui et al., adopted *software symbiotes* [88], an on-device binary instrumentation to automatically insert hooks in embedded firmwares. Their solution allows to insert pieces of code that can be used to interact with the original firmware. How-

ever, while this allows some analysis (like tracing), performing advanced dynamic analysis often requires to be able to run the firmware code inside an emulator.

Dynamic analysis based on virtualization has already been proposed in the past [166], also in embedded systems contexts [171, 133]. However, *Avatar* aims at overcoming many of the limitations of pure-virtualization systems, by providing an hybrid system where code execution can be transferred back and forth between the device and an emulator, as well as a full framework to orchestrate all the analysis steps.

The state migration technique employed by *Avatar* is highly influenced by existing solutions been used to improve the performance during hot-migration of virtual machines. In particular, our approach is a simplified version of the one proposed by Clark et al. [73], where *Avatar* is the arbiter of a *managed migration*, which can either happen in a single *stop-and-copy phase* (as in full-separation mode) or in an event-driven *pull-phase* (during context switching).

The "security by obscurity" approach is still relevant among embedded systems manufacturers and has lead in the past to the discovery of major weaknesses in commonly deployed technologies [201]. We believe that *Avatar* represents a flexible solution to provide a symbolic analysis environment which can greatly speed-up such blackbox analysis cases, aiming at automatically reverse engineer input formats [89, 52] and detect hidden data structures [229]. In the past, backdoors and insecure firmware update facilities were found into embedded systems, often disguised into other standard interfaces such as Printer Job Language updates for HP printers [84]. In our experiments we showed how *Avatar* can be used to actively look for such backdoors, by symbolically executing input parsing routines.

Davidson et al. [92] present a tool to perform symbolic execution of embedded firmware for MSP430-based devices. Like *Avatar*, this tool is based on the KLEE symbolic execution engine. However, it relies on firmware's source code as well as on documented SoCs, peripherals mapping, or on a simple emulation layer for them, all of those are rarely available for commercial devices.

Delugre [95] reports on the techniques that were used to reverse engineer the firmware of a PCI network card, and to develop a backdoored firmware. For this purpose, QEMU was adapted to emulate the firmware and to forward IO access to the device. However, this was limited by bad performance. We have seen similar performance blockers when using *Avatar* in full separation mode, but the ability to perform memory optimization and push back code to the physical device allow *Avatar* to overcome such limitations.

Dedicated hardware support can provide a very good solution to improve efficiency of debugging, improving significantly the ability to replay events and system status. In [258] Xu et al., presents an hardware architecture for recording precise events and replay them during debugging sessions. For this purpose custom hardware logs memory and taps on several important internal features (e.g., cache lines). Simpler systems also exist, like In-Circuit Emulators [255], which replace the CPU core by an emulated CPU which can then directly interact with

hardware peripherals. While *Avatar* could make use of such features, it also aims at enabling analysis on devices without such dedicated hardware support.

2.8 Conclusion

This paper introduced *Avatar*, a new framework for dynamic analysis of embedded devices' firmwares. *Avatar* enables the execution of firmware code in an analysis-friendly emulator by forwarding memory access to the real device. This allows to analyze firmwares that rely on completely unknown peripherals.

Avatar proved to be capable of acceptable performances and flexibility in three real-word tests, performed on a variety of target devices and with different goals. It was successfully used across these three scenarios, which included a common reverse engineering task, a vulnerability discovery and a hardcoded backdoor detection.

Future work will consist in integrating better analysis techniques with avatar to improve its bug detection rate. For example, augmenting *Avatar* with techniques like those used in Howard [227] would allow to recover memory structures and therefore improve bug detection, while other techniques as used in AEG [26] could be applicable as well. Another area where significant improvements can be achieved is in providing improved state exploration heuristics, that lead to better coverage or to the analysis of more error prone code [132].

Finally, *Avatar* has been tested on ARM embedded systems and could easily support x86 targets, but could be ported with reasonable effort to a wider set of architectures supported by QEMU such as MIPS and PowerPC, in order to analyze many other devices.

3

A Large-Scale Analysis of the Security of Embedded Firmwares

Preamble: Relation to the Research Roadmap

The security of embedded devices in general, and of the Internet of Things in particular, is an emerging topic that is attracting a lot of attention in the security community. The final Report on Threats on the Future Internet and Research Roadmap (D4.4) lists both of them as important research directions.

The main problems that limited previous attempts to study this field are the lack of a common dataset to perform the experiments, and the intrinsic diversity of the devices architectures and data formats. The following paper presents the first large scale study of the security of embedded devices, emphasizing both the advantages and the problems that arise when performing this kind of experiments:

Andrei Costin, Jonas Zaddach, Aurlien Francillon, Davide Balzarotti "A Large Scale Analysis of the Security of Embedded Firmwares" Proceedings of the 23rd USENIX Security Symposium (USENIX Security) – August 2014

Abstract

As embedded systems are more than ever present in our society, their security is becoming an increasingly important issue. However, based on the results of many recent analyses of individual firmware images, embedded systems acquired a reputation of being insecure. Despite these facts, we still lack a global understanding of embedded systems' security as well as the tools and techniques needed to support such general claims.

In this paper we present the first public, large-scale analysis of firmware images. In particular, we unpacked 32 thousand firmware images into 1.7 million individual files, which we then statically analyzed. We leverage this large-scale analysis to bring new insights on the security of embedded devices and to underline and detail several important challenges that need to be addressed in future research. We also show the main benefits of looking at many different devices at the same time and of linking our results with other large-scale datasets such as the ZMap's HTTPS survey.

In summary, without performing sophisticated static analysis, we discovered a total of 38 previously unknown vulnerabilities in over 693 firmware images. Moreover, by correlating similar files inside apparently unrelated firmware images, we were able to extend some of those vulnerabilities to over 123 different products. We also confirmed that some of these vulnerabilities altogether are affecting at least $140K$ devices accessible over the Internet. It would not have been possible to achieve these results without an analysis at such wide scale.

We believe that this project, which we plan to provide as a firmware unpacking and analysis web service[1], will help shed some light on the security of embedded devices.

3.1 Introduction

Embedded systems are omnipresent in our everyday life. For example, they are the core of various Common-Off-The-Shelf (COTS) devices such as printers, mobile phones, home routers, and computer components and peripherals. They are also present in many devices that are less consumer oriented such as video surveillance systems, medical implants, car elements, SCADA and PLC devices, and basically anything we normally call *electronics*. The emerging phenomenon of the Internet-of-Things (IoT) will make them even more widespread and interconnected.

All these systems run special software, often called *firmware*, which is usually distributed by vendors as *firmware images* or *firmware updates*. Several definitions for *firmware* exist in the literature. The term was originally introduced to describe the CPU microcode that existed "somewhere" between the hardware and the software layers. However, the word quickly assumed a broader meaning, and the IEEE

[1] http://firmware.re

Std 610.12-1990 [10] extended the definition to cover the *"combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device"*.

Nowadays, the term *firmware* is more generally used to describe the software that is embedded in a hardware device. Like traditional software, embedded devices' firmware may have bugs or misconfigurations that can result in vulnerabilities for the devices which run that particular code. Due to anecdotal evidence, embedded systems acquired a bad security reputation, generally based on case by case experiences of failures. For instance, a car model throttle control fails [140] or can be maliciously taken over [64, 162]; a home wireless router is found to have a backdoor [143, 15, 137], just to name a few recent examples. On the one hand, apart from a few works that targeted specific devices or software versions [126, 85, 217], to date there is still no large-scale security analysis of firmware images. On the other hand, manual security analysis of firmware images yields very accurate results, but it is extremely slow and does not scale well for a large and heterogeneous dataset of firmware images. As useful as such individual reports are for a particular device or firmware version, these alone do not allow to establish a general judgment on the overall state of the security of firmware images. Even worse, the same vulnerability may be present in different devices, which are left vulnerable until those flaws are re-discovered independently by other researchers [143]. This is often the case when several *integration vendors* rely on the same subcontractors, tools, or SDKs provided by *development vendors*. Devices may also be branded under different names but may actually run either the same or similar firmware. Such devices will often be affected by exactly the same vulnerabilities, however, without a detailed knowledge of the internal relationships between those vendors, it is often impossible to identify such similarities. As a consequence, some devices will often be left affected by known vulnerabilities even if an updated firmware is available.

3.1.1 Methodology

Performing a large-scale study of the security of embedded devices by actually running the physical devices (i.e., using a dynamic analysis approach) has several major drawbacks. First of all, physically acquiring thousands of devices to study would be prohibitively expensive. Moreover, some of them may be hard to operate outside the system for which they are designed — e.g., a throttle control outside a car. Another option is to analyze existing online devices as presented by Cui and Stolfo [87]. However, some vulnerabilities are hard to find by just looking at the running device, and it is ethically questionable to perform any nontrivial analysis on an online system without authorization.

Unsurprisingly, static analysis scales better than dynamic analysis as it does not require access to the physical devices. Hence, we decided to follow this approach in our study. Our methodology consists of collecting firmware images for as many devices and vendors as possible. This task is complicated by the fact that

firmware images are diverse and it is often difficult to tell firmware images apart from other files. In particular, distribution channels, packaging formats, installation procedures, and availability of meta-data often depend on the vendor and on the device type. We then designed and implemented a distributed architecture to unpack and run simple static analysis tasks on the collected firmware images. However, the contribution of this paper is not in the static analysis techniques we use (for example, we did not perform any static *code* analysis), but to show the advantages of an horizontal, large-scale exploration. For this reason, we implemented a correlation engine to compare and find similarities between all the objects in our dataset. This allowed us to quickly "propagate" vulnerabilities from known vulnerable devices to other systems that were previously not known to be affected by the same vulnerability.

Most of the steps performed by our system are conceptually simple and could be easily performed manually on a few devices. However, we identified *five major challenges* that researchers need to address in order to perform large scale experiments on thousands of different firmware images. These include the problem of building a representative dataset (Challenge A in Section 3.2), of properly identifying individual firmware images (Challenge B in Section 3.2), of unpacking custom archive formats (Challenge C in Section 3.2), of limiting the required computation resources (Challenge D in Section 3.2), and finally of finding an automated way to confirm the results of the analysis (Challenge E in Section 3.2). While in this paper we do not propose a complete solution for all these challenges, we discuss the way and the extent to which we dealt with some of these challenges to perform a systematic, automated, large-scale analysis of firmware images.

3.1.2 Results Overview

For our experiments we collected an initial set of `759,273` files (totaling `1.8TB` of storage space) from publicly accessible firmware update sites. After filtering out the obvious noise, we were left with `172,751` potential firmware images. We then sampled a set of `32,356` firmware candidates that we analyzed using a private cloud deployment of `90` worker nodes. The analysis and reports resulted in a `10GB` database.

The analysis of sampled files led us to automatically discover and report 38 new vulnerabilities (fixes for some of these are still pending) and to confirm several that were already known [137, 143]. Some of our findings include:

- We extracted private RSA keys and their self-signed certificates used in about 35,000 online devices (mainly associated with surveillance cameras).
- We extracted several dozens of hard-coded password hashes. Most of them were weak, and therefore we were able to easily recover the original passwords.
- We identified a number of possible backdoors such as the `authorized_keys` file (which lists the SSH keys that are allowed to remotely connect

to the system), a number of hard-coded `telnetd` credentials affecting at least 2K devices, hard-coded web-login admin credentials affecting at least 101K devices, and a number of backdoored daemons and web pages in the web-interface of the devices.

- Whenever a new vulnerability was discovered (by other researchers or by us) our analysis infrastructure allowed us to quickly find related devices or firmware versions that were likely affected by the same vulnerability. For example, our *correlation techniques* allowed us to correctly extend the list of affected devices for variations of a `telnetd` hard-coded credentials vulnerability. In other cases, this led us to find a vulnerability's root problem spread across multiple vendors.

3.1.3 Contributions

In summary this paper makes the following contributions:

- We show the advantages of performing a large-scale analysis of firmware images and describe the main challenges associated with this activity.
- We propose a framework to perform firmware collection, filtering, unpacking and analysis at large scale.
- We implemented several efficient static techniques that we ran on $32,356$ firmware candidates.
- We present a correlation technique which allows to propagate vulnerability information to similar firmware images.
- We discovered 693 firmware images affected by at least one vulnerability and reported 38 new CVEs.

3.2 Challenges

As mentioned in the previous section, there are clear advantages of performing a wide-scale analysis of embedded firmware images. In fact, as is often the case in system security, certain phenomena can only be observed by looking at the global picture and not by studying a single device (or a single family of devices) at a time.

However, large-scale experiments require automated techniques to obtain firmware images, unpack them, and analyze the extracted files. While these are easy tasks for a human, they become challenging when they need to be fully automated. In this section we summarize the five main challenges that we faced during the design and implementation of our experiments.

Challenge A: Building a Representative Dataset

The embedded systems environment is heterogeneous, spanning a variety of devices, vendors, architectures, instruction sets, operating systems, and custom components. This makes the task of compiling a *representative* and *balanced* dataset of firmware images a difficult problem to solve.

The real market distribution of a certain hardware architecture is often unknown, and it is hard to compare different classes of devices (e.g., medical implants vs. surveillance cameras). Which of them need to be taken into account to build a representative firmware dataset? How easy is it to generalize a technique that has only been tested on a certain brand of routers to other vendors? How easy is it to apply the same technique to other classes of devices such as TVs, cameras, insulin pumps, or power plant controllers?

From a practical point of view, the lack of centralized points of collection (such as the ones provided by antivirus vendors or public sandboxes in the malware analysis field) makes it difficult for researchers to gather a large and well triaged dataset. Firmware often needs to be downloaded from the vendor web pages, and it is not always simple, even for a human, to tell whether or not two firmware images are for the same physical device.

Challenge B: Firmware Identification

One challenge often encountered in firmware analysis and reverse engineering is the difficulty of reliably extracting meta-data from a firmware image. For instance, such meta-data includes the vendor, the device product code and purpose, the firmware version, and the processor architecture, among many other details.

In practice, the diversity of firmware file formats makes it harder to even recognize that a given file downloaded from a vendor website is a firmware at all. Often firmware updates come in unexpected formats such as *HP Printer Job Language* and *PostScript* documents for printers [78, 77, 85], *DOS executables* for BIOS, and *ISO images* for hard disk drives [265].

In many cases, the only source of reliable information is the official vendor documentation. While this is not a problem when looking manually at a few devices, extending the analysis to hundreds of vendors and thousands of firmware images automatically downloaded from the Internet is challenging. In fact, the information retrieval process is hard to automate and is error prone, in particular for certain classes of meta-data. For instance, we often found it hard to infer the correct version number. This makes it difficult for a large-scale collection and analysis system to tell which is the latest version available for a certain device, and even if two firmware images corresponded to different versions for the same device. This further complicates the task of building an unbiased dataset.

Challenge C: Unpacking and Custom Formats

Assuming the analyst succeeded in collecting a representative and well labeled dataset of firmware images, the next challenge consists in locating and extracting important functional blocks (e.g., binary code, configuration files, scripts, web interfaces) on which static analysis routines can be performed.

While this task would be easy to address for traditional software components, where standardized formats for the distribution of machine code (e.g., PE and ELF), resources (e.g., JPEG and GZIP) and groups of files (e.g., ZIP and TAR) exist, embedded software distribution lacks standards. Vendors have developed their own file formats to describe flash and memory images. In some cases those formats are compressed with non-standard compression algorithms. In other cases those formats are obfuscated or encrypted to prevent analysis. Monolithic firmware, in which the bootloader, the operating system kernel, the applications, and other resources are combined together in a single memory image are especially challenging to unpack.

Forensic strategies, like file *carving*, can help to extract known file formats from a binary blob. Unfortunately those methods have drawbacks: On the one hand, they are often too aggressive with the result of extracting data that matches a file pattern only by chance. On the other hand, they are computationally expensive, since each unpacker has to be tried for each file offset of the binary firmware blob.

Finally, if a binary file has been extracted that does not match any known file pattern, it is impossible to say if this file is a data file, or just another container format that is not recognized by the unpacker. In general, we tried to unpack at least until reaching uncompressed files. In some cases, our extraction goes one step further and tries to extract sections, resources and compressed streams (e.g., for the ELF file format).

Challenge D: Scalability and Computational Limits

One of the main advantages of performing a wide-scale analysis is the ability of correlating information across multiple devices. For example, this allowed us to automatically identify the re-use of vulnerable components among different firmware images, even from different vendors.

Capturing the global picture of the relationship between firmware images would require the one-to-one comparison of each pair of unpacked files. Fuzzy hashes (such as sdhash [214] and ssdeep [161]) are a common and effective solution for this type of task and they have been successfully used in similar domains, e.g., to correlate samples that belong to the same malware families [101, 36]. However, as described in more detail in Section 3.3.4, computing the similarity between the objects extracted from 26,275 firmware images requires 10^{12} comparisons. Using the simpler fuzzy hash variant, we estimate that on a single dual-core computer this

task would take approximately 850 days[2]. This simple estimation highlights one of the possible *computational challenges* associated with a large-scale firmware analysis. Even if we had a perfect database design and a highly optimized in-memory database, it would still be hard to compute, store, and query the fuzzy hash scores of all pairs of unpacked files. A distributed computational infrastructure can help reduce the total time since the task itself is parallelizable [184]. However, since the number of comparisons grows quadratically with the number of elements to compare, this problem quickly becomes impracticable for large image datasets. If, for example, one would like to build a fuzzy hash database for our whole dataset, which is just five times the size of the current sampled dataset, this effort would already take more than 150 CPU years instead of 850 CPU days. Our attempt to use the GPU-assisted fuzzy hashing provided by `sdhash` [214] only resulted in a limited speedup that was not sufficient to perform a full-scale comparison of all files in our dataset.

Challenge E: Results Confirmation

The first four challenges were mostly related to the collection of the dataset and the pre-processing of the firmware images. Once the code or the resources used by the embedded device have been successfully extracted and identified, researchers can focus their attention on the static analysis. Even though the details and goals of this step are beyond the scope of this paper, in Section 3.3.3 we present some examples of simple static analysis and we discuss the advantages of performing these techniques on a large scale.

However, one important research challenge remains regarding the way the results of static analysis can be confirmed. For example, we can consider a scenario where a researcher applies a new vulnerability detection technique to several thousand firmware images. Those images were designed to run on specific embedded devices, most of which are not available to the researcher and would be hard and costly to acquire. Lacking the proper hardware platform, there is still no way to manually or automatically test the *affected code* to confirm or deny the findings of the static analysis.

For example, in our experiments we identified a firmware image that included the `PHP 5.2.12` banner string. This allowed us to easily identify several vulnerabilities associated with that version of the PHP interpreter. However, this is insufficient to determine if the PHP interpreter is vulnerable, since the vendor may have applied patches to correct known vulnerabilities without this being reflected in the version string. In addition, the vendor might have used an architecture and/or a set of compilation options which produced a non-vulnerable build of the component. Unfortunately, even if a proof of concept attack exists for that vulnerability, without the proper hardware it is impossible to test the firmware and confirm or deny the presence of the problem.

[2] This is mainly because comparing fuzzy hashes is not a simple bit string comparison but actually involves a rather complex algorithm and high computational effort.

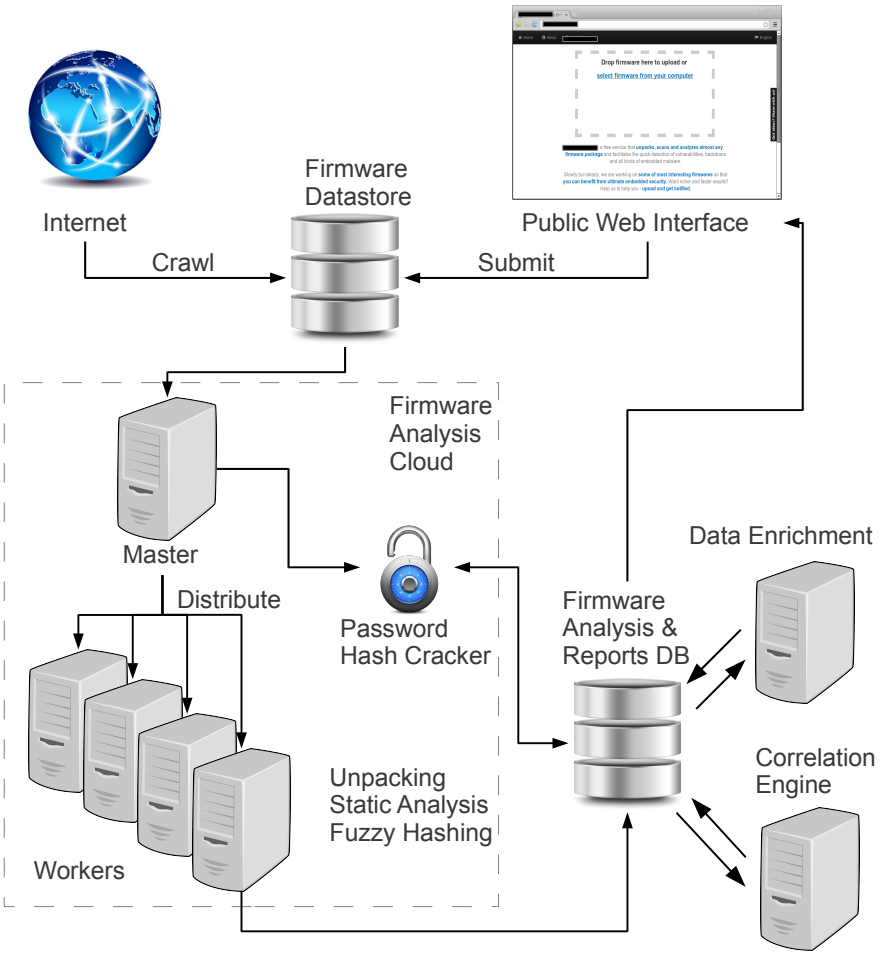


Fig. 3.1: Architecture of the entire system.

Confirming the results of the static analysis on firmware devices is a tedious task requiring manual intervention from an expert. Scaling this effort to thousands of firmware images is even harder. Therefore, we believe the development of new techniques is required to accurately deal with this problem at a large scale.

3.3 Setup

In this section we first present the design of our distributed static analysis and correlation system. Then we detail the techniques we used, and how we addressed the challenges described in Section 3.2.

3.3.1 Architecture

Figure 3.1 presents an overview of our architecture. The first component of our analysis platform is the *firmware data store*, which stores the unmodified firmware files that have been retrieved either by the *web crawler* or that have been submitted through the public *web interface*. When a new file is received by the firmware data store, it is automatically scheduled to be processed by the *analysis cloud*. The analysis cloud consists of a master node, and a number of worker and hash cracking nodes. The *master node* distributes unpacking jobs to the *worker nodes* (Figure 3.2), which unpack and analyze firmware images. *Hash cracking nodes* process password hashes that have been found during the analysis, and try to find the corresponding plaintext passwords. Apart from coordinating the worker nodes, the master node also runs the *correlation engine* and the *data enrichment system* modules. These modules improve the reports with results from the cross-firmware analysis.

The analysis cloud is where the actual analysis of the firmware takes place. Each firmware image is first submitted to the *master node*. Subsequently, *worker nodes* are responsible for unpacking and analyzing the firmware and for returning the results of the analysis back to the master node. At this point, the master node will submit this information to the *reports database*. If there were any uncracked password hashes in the analyzed firmware, it will additionally submit those hashes to one of the *hash cracking nodes* which will try to recover the plaintext passwords.

It is important to note that only the results of the analysis and the meta-data of the unpacked files are stored in the database. Even though we do not currently use the extracted files after the analysis, we still archive them for future work, or in case we want to review or enhance a specific set of analyzed firmware images.

The architecture contains two other components: the *correlation engine* and the *data enrichment system*. Both of them fetch the results of the firmware analysis from the reports database and perform additional tasks. The correlation engine identifies a number of "interesting" files and tries to correlate them with any other file present in the database. The enrichment system is responsible for enhancing the information about each firmware image by performing online scans and lookup queries (e.g., detecting vendor name, device name/code and device category).

In the remainder of this section we describe each step of the firmware analysis in more detail so that our experiments can be reproduced.

3.3.2 Firmware Acquisition and Storage

The first step of our experiments consisted in gathering a firmware collection for analysis. We achieved this goal by using mainly two methods: a web crawler that automatically downloads files from manufacturers' websites and specialized mirror sites, and a website with a submission interface where users can submit firmware images for analysis.

We initialized the crawler with tens of support pages from well known manufacturers such as Xerox, Bosch, Philips, D-Link, Samsung, LG, Belkin, etc. Sec-

ond, we used public FTP indexing engines [3] to search for files with keywords related to firmware images (e.g., `firmware`). The result of such searches yields either directory URLs, which are added to the crawler list of URLs to index and download, or file URLs, which are directly downloaded by the crawler. At the same time, the script strips filenames out of the URLs to create additional directory URLs.

Finally, we used Google Custom Search Engines (GCSE) [6] to create customized search engines. GCSE provides a flexible API to perform advanced search queries and returns results in a structured way. It also allows to programmatically create a very customized CSE on-the-fly using a combination of RESTful and XML APIs. For example, a CSE is created using `support.nikonusa.com` as the "Sites to Search" parameter. Then a firmware related query is used on the CSE such as ``` ``firmware download'' ```. The CSE from the above example returns 2,210 results at the time of this publication. The result URLs along with associated meta-data are retrieved via the JSON API. Each URL was then used by the crawler or as part of other dynamic CSE, as previously described. This allowed us to mine additional firmware images and firmware repositories.

We chose not to filter data at collection time, but to download files greedily, deciding at a later stage if the collected files were firmware images or not. The reason for this decision is two-fold. First, accompanying files such as manuals and user guides can be useful for finding additional download locations or for extracting contained information (e.g., model, default passwords, update URLs). Second, as we mentioned previously, it is often difficult to distinguish firmware images from other files. For this reason, filtering a large dataset is better than taking a chance to miss firmware files during the downloading phase. In total, we crawled 284 sites and stopped downloading once the collection of files reached 1.8TB of storage. The actual storage required for this amount of data is at least 3-4 times larger, since we used mirrored backup storage, as well as space for keeping the unpacked files and files generated during the unpacking (e.g., logs and analysis results).

The public *web submission interface* provides a means for security researchers to submit firmware files for analysis. After the analysis is completed, the platform produces a report with information about the firmware contents as well as similarities to other firmware in our database. We have already received tens of firmware images through the submission interface. While this is currently a marginal source of firmware files, we expect that more firmware will be submitted as we advertise our service. This will also be a unique chance to have access to firmware images that are not generally available and, for example, need to be manually extracted from a device.

[3] FTP indexing engines such as: `www.mmnt.ru`, `www.filemare.com`, `www.filewatcher.com`, `www.filesearching.com`, `www.ftpsearch.net`, `www.search-ftps.com`

Files fetched by the web crawler and received from the web submission interface are added to the *firmware data store*. Files are simply stored on a file system and a database is used for meta-data (e.g., file checksum, size, download location).

3.3.3 Unpacking and Analysis

The next step towards the analysis of a firmware image is to unpack and extract the contained files or objects. The output of this phase largely depends on the type of firmware. In some examples, executable code and resources (such as graphics files or HTML code) can be linked into a binary blob that is designed to be directly copied into memory by a bootloader and then executed. Some other firmware images are distributed in a compressed and obfuscated file which contains a block-by-block copy of a flash image. Such an image may consist of several partitions containing a bootloader, a kernel and a file system.

Unpacking Frameworks

There are three main tools to unpack arbitrary firmware images: binwalk [3], FRAK [83] and Binary Analysis Toolkit (BAT) [241].

Binwalk is a well known firmware unpacking tool developed by Craig Heffner [3]. It uses pattern matching to locate and carve files from a binary blob. Additionally, it also extracts meta-data such as license strings.

FRAK is an unpacking toolkit first presented by Cui et al. [85]. Even though the authors mention that the tool would be made publicly available, we were not able to obtain a copy. We therefore had to evaluate its unpacking performance based on the device vendors and models that FRAK supports, according to [85]. We estimated that FRAK would have unpacked less than 1% of the files we analyzed, while our platform was able to unpack more than 81% of them. This said, both would be complementary as some of the file formats FRAK unpacks are unsupported by our tool at present.

The Binary Analysis Toolkit (BAT), formerly known as GPLtool, was originally designed by Tjaldur software to detect GPL violations [138, 241]. To this end, it recursively extracts files from a firmware blob and matches strings with a database of known strings from GPL projects. Additionally, BAT supports file carving similar to binwalk.

Table 3.1 shows a simple comparison of the unpacking performance of each framework on a few samples of firmware images. We chose to use BAT because it is the most complete tool available for our purposes. It also has a significantly lower rate of false positive extractions compared to binwalk. In addition, binwalk did not support recursive unpacking at the time when we decided on an unpacking framework. Nevertheless, the interface between our framework and BAT has been designed to be generic so that integrating other unpacking toolkits (such as binwalk) is easy.

We developed a range of additional plugins for BAT. These include plugins which extract interesting strings (e.g., software versions or password hashes), add

Table 3.1: Comparison of Binwalk, BAT, FRAK and our framework. The last three columns show if the respective unpacker was able to extract the firmware. Note that this is a non statistically significant sample which is given for illustrating unpacking performance (manual analysis of each firmware is time consuming). As FRAK was not available for testing, its unpacking performance was estimated based on information from [83]. The additional performance of our framework stems from the many customizations we have incrementally developed over BAT (Figure 3.2).

Device	Vendor	OS	Binwalk	BAT	FRAK	Our framework
PC	Intel	BIOS	✗	✗	✗	✗
Camera	STL	Linux	✗	✓	✗	✓
Router	Bintec	-	✗	✗	✗	✗
ADSL Gateway	Zyxel	ZynOS	✓	✓	✗	✓
PLC	Siemens	-	✓	✓	✗	✓
DSLAM	-	-	✓	✓	✗	✓
PC	Intel	BIOS	✓	✓	✗	✓
ISDN Server	Planet	-	✓	✓	✗	✓
Voip	Asotel	Vxworks	✓	✓	✗	✓
Modem	-	-	✗	✗	✗	✓
Home Automation	Belkin	Linux	✗	✗	✗	✓
			55%	64%	0%	82%

unpacking methods, gather statistics and collect interesting files such as private key files or `authorized_keys` files. In total we added 35 plugins to the existing framework.

Password Hash Cracking

Password hashes found during the analysis phase are passed to a hash cracking node. These nodes are dedicated physical hosts with a Nvidia Tesla GPU [173] that run a CUDA-enabled [202] version of *John The Ripper* [203]. John The Ripper is capable of brute forcing most encoded password hashes and detecting the type of hash and salt used. In addition to this, a dictionary can be provided to seed the password cracking. For each brute force attempt, we provide a dictionary built from common password lists and strings extracted from firmwares, manuals, *readme* files and other resources. This allows to find both passwords that are directly present in those files as well as passwords that are weak and based on keywords related to the product.

Parallelizing the Unpacking and Analysis

To accelerate the unpacking process, we distributed this task on several worker nodes. Our distributed environment is based on the *distributed-python-for-scripting* framework [237]. Data is synchronized between the repository and the nodes using *rsync (over ssh)* [243].

Our loosely coupled architecture allows us to run worker nodes virtually anywhere. For instance, we instantiated worker virtual machines on a local VMware

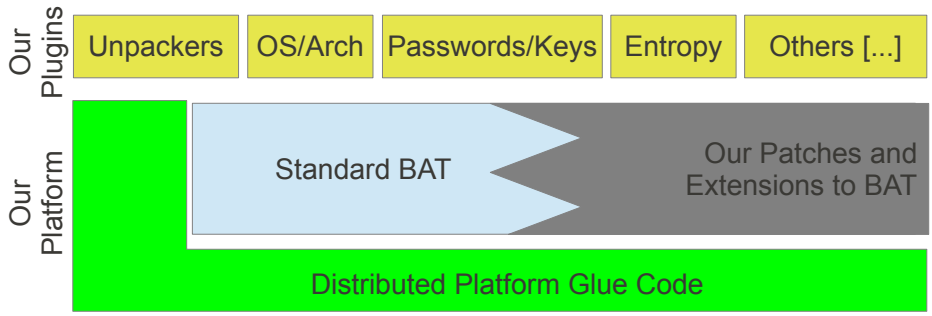


Fig. 3.2: Architecture of a single worker node.

server and several OpenStack servers, as well as on Amazon EC2 instances. At the time of this publication we were using 90 such virtual machines to analyze firmware files.

3.3.4 Correlation Engine

The unpacked firmware images and analysis results are stored into the *analysis & reports database*. This allows us to perform queries, to generate reports and statistics, and to easily integrate our results with other external components. The correlation engine is designed to find similarities between different firmware images. In particular, the comparison is made along four different dimensions: shared credentials, shared self-signed certificates, common keywords, and fuzzy hashes of the firmwares and objects within the firmwares.

Shared Credentials and Self-Signed Certificates

Shared credentials (such as hard coded *non-trivial* passwords) and shared self-signed certificates are effective in finding strong connections between different firmware images of the same vendor, or even firmwares of different vendors. For example, we were able to correlate two brands of CCTV systems based on a common *non-trivial* default password.

Therefore, finding a password of one vendor's product can directly impact the security of others. We also found a similar type of correlation for two other CCTV vendors that we linked through the same self-signed certificate, as explained in Section 3.5.2.

Keywords

Keywords correlation is based on specific strings extracted by our static analysis plugins. In some cases, for example in Section 3.5.1, the keyword "`backdoor`" revealed several other keywords. By using the extended set of keywords we clustered several vendors prone to the same backdoor functionality, possibly affecting

$500,000$ devices. In other cases, files inside firmware images contain compilation and SDK paths. This turns out to be sufficient to cluster firmware images of different devices.

Fuzzy hashes

Fuzzy hash triage (comparison, correlation and clustering) is the most generic correlation technique used by our framework. The engine computes both the *ssdeep* and the *sdhash* of every single object extracted from the firmware image during the unpacking phase. This is a powerful technique that allows us to find files that are "similar" but for which a traditional hash (such as *MD5* or *SHA1*) would not match. Unfortunately, as we already mentioned in Section 3.2, a complete one-to-one comparison of fuzzy hashes is currently infeasible on a large scale. Therefore, we compute the fuzzy hashes of each file that was successfully extracted from a firmware image and store this result. When a file is found to be interesting we perform the fuzzy hash comparison between this file's hash and all stored hashes.

For example, a file (or all files unpacked from a firmware) may be flagged as interesting because it is affected by a known vulnerability, or because we found it to be vulnerable by static analysis. If another firmware contains a file that is similar to a file from a vulnerable firmware, then there might be a chance that the first firmware is also vulnerable. We present such an example in Section 3.5.3, where this approach was successful and allowed us to propagate known vulnerabilities of one device to other similar devices of *different* vendors.

Future work

In the literature, there are several approaches proposed to perform comparison, clustering, and triage on a large scale. Jang et al. propose large-scale triage techniques of PC malware in BitShred [147]. The authors concluded that at the rate of 8,000 unique malware samples per day, which required 31M comparisons, it is unfeasible on a single CPU to perform one-to-one comparisons to find malware families using hierarchical clustering. French and Casey [34] propose, before fuzzy hash comparison, to perform a "bins" partitioning approach based on the block and file sizes. This approach, for their particular dataset and bins partitioning strategy, allowed on average to reduce the search space for a given fuzzy hash down to 16.9%. Chakradeo et al. [62] propose MAST, an effective and well performing triage architecture for mobile market applications. It solves the manual and resource-intensive automated analysis at market-scale using Multiple Correspondence Analysis (MCA) statistical method.

As a future work, there are several possible improvements to our approach. For instance, instead of performing all comparisons on a single machine, we could adopt a distributed comparison and clustering infrastructure, such as the Hadoop implementation of MapReduce [94] used by BitShred. Second, on each comparison

and clustering node we could use the "bins" partitioning approach from French and Casey [34].

3.3.5 Data Enrichment

The data enrichment phase is responsible for extending the knowledge base about firmware images, for example by performing automated queries and passive scans over the Internet. In the current prototype, the data enrichment relies on two simple techniques. First, it uses the `<title>` tag of web pages and `authentication realms` of web servers when these are detected inside a firmware. This information is then used to build targeted *search queries* (such as *"intitle:Router ABC-123 Admin Page"*) for both Shodan [9] and GCSE.

Second, we correlate SSL certificates extracted from firmware images to those collected by the ZMap project. ZMap was used in [103] to scan the whole IPv4 address space on the 443 port, collecting SSL certificates in a large database.

Correlating these two large-scale databases (i.e., ZMap's HTTPS survey and our firmware database) provides new insights. For example, we are able to quickly evaluate the severity of a particular vulnerability by identifying publicly reachable devices that are running a given firmware image. This gives a good estimate for the number of publicly accessible vulnerable devices.

For instance, our framework found 41 certificates having unprotected private keys. Those keys were extracted from firmware images in the unpacking and analysis phase. The data enrichment engine subsequently found the same self-signed certificate in over 35K devices reachable on the Internet. We detail this case study in Section 3.5.2.

3.3.6 Setup Development Effort

Our framework relies on many existing tools. In addition to this, we have put a considerable effort (over 20k lines of code according to `sloccount` [254]) to extend BAT, develop new unpackers, create the results analysis platform and run results interpretation.

3.4 Dataset and Results

In this section we describe our dataset and we present the results of the global analysis, including the discussion of the new vulnerabilities and the common bad practices we discovered in our experiments. In Section 3.5, we will then present a few concrete case studies, illustrating how such a large dataset can provide new insights into the security of embedded systems.

3.4.1 General Dataset Statistics

While we currently collect firmware images from multiple sources, most of the images in our dataset have been downloaded by crawling the Internet. As a

consequence, our dataset is biased towards devices for which firmware updates can be found online, and towards known vendors that maintain well organized websites.

We also decided to exclude firmware images of smartphones from our study. In fact, popular smartphone firmware images are complete operating system distributions, most of them iOS, Android or Windows based – making them closer to general purpose systems than to embedded devices.

Our crawler collected 759,273 files, for a total of 1.8TB of data. After filtering out the files that were clearly unrelated (e.g., manuals, user guides, web pages, empty files) we obtained a dataset of 172,751 files. Our architecture is constantly running to fetch more samples and analyze them in a distributed fashion. At the time of this publication the system was able to process (unpack and analyze) 32,356 firmware images.

Firmware Identification The problem of properly identifying a firmware image (Challenge 2) still requires a considerable amount of manual effort. Doing so accurately and automatically at a large scale is a daunting task. Nevertheless, we are interested in having an estimate of the number of actual firmware images in our dataset.

For this purpose we manually analyzed a number of random samples from our dataset of 172,751 potential firmware images and computed a *confidence interval* [50] to estimate the global representativeness in the dataset. In particular, after manually analyzing 130 random files from the total of 172,751, we were able to mark only 44 as firmware images. This translates to a proportion of 34% ($\pm$ 8%) firmware images on our dataset – with a 95% confidence. The manual analysis process took approximately one person-week because the inspection of the extracted files for firmware code is quite tedious.

We can therefore expect our dataset to contain between 44,431 and 72,520 firmware images (by applying $34\%-8\%$, and $34\%+8\%$ respectively, to the entire candidates set of 172,751). While the range is still relatively large, this estimation gives a 95% reliable measure of the useful data in our sample. We also developed a heuristic to automatically detect if a file is successfully unpacked or not. This heuristic takes multiple parameters, such as the number, type and size of files carved out from a firmware, into account. Such an empirical heuristic is not perfect, but it can guide our framework to mark a file as unpacked or not, and then take actions accordingly.

Files Analysis As described in Section 3.3.3, unpacking unknown files is an error-prone and time-consuming task. In fact, when the file format is not recognized, unpacking relies on a slow and imprecise carving approach. File carving is essentially an attempt to unpack at every offset of the file, iterating over several known signatures (e.g., archive magic headers).

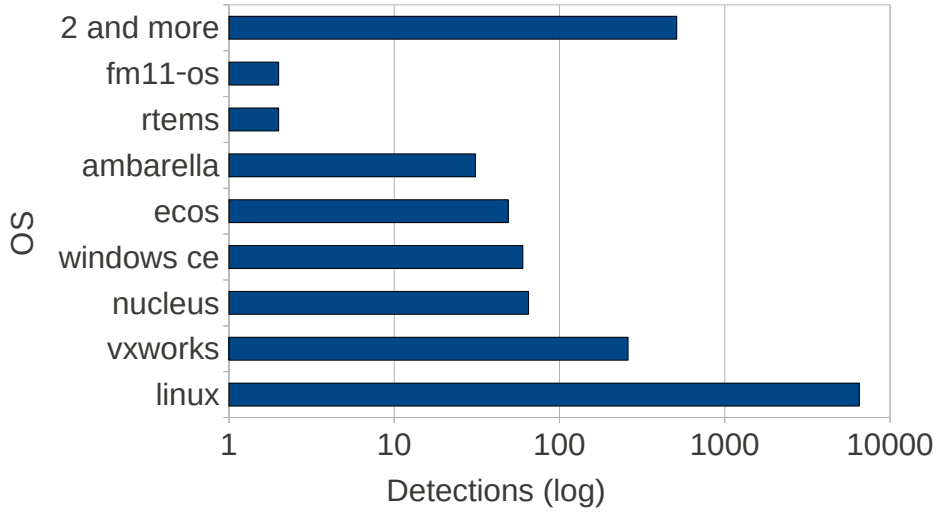


Fig. 3.3: OS distribution among firmware images.

As a result, out of the 32,356 files we processed so far, 26,275 were successfully unpacked. The process is nevertheless continuous and more firmware images are being unpacked over time.

3.4.2 Results Overview

In the rest of the section we present the results of the analysis performed by our plugins right after each firmware image was unpacked.

Files Formats The majority of initial files being unpacked were identified as *compressed files* or *raw data*. Once unpacked, most of those firmware images were identified as targeting ARM (63%) devices, followed by MIPS (7%). As reported in Figure 3.3, Linux is the most frequently encountered embedded operating system in our dataset – being present in more than three quarters (86%) of all analyzed firmware images. The remaining images contain proprietary operating systems like VxWorks, Nucleus RTOS and Windows CE, which altogether represent around 7%. Among Linux based firmware images, we identified 112 distinct Linux kernel versions.

Password Hashes Statistics Files like `/etc/passwd` and `/etc/shadow` store hashed versions of account credentials. These are usual targets for attackers since they can be used to retrieve passwords which often allow to login remotely to a device at a later time. Hence, an analysis of these files can help understanding how well an embedded device is protected.

Our plugin responsible for collecting entries from `/etc/passwd` and `/etc/shadow` files retrieved 100 distinct password hashes, covering 681 distinct firm-

ware images and belonging to 27 vendors. We were also able to recover the plaintext passwords for 58 of those hashes, which occur in 538 distinct firmware images. The most popular passwords were `<empty>`, `pass`, `logout`, and `helpme`. While these may look trivial, it is important to stress that they are actually used in a large number of embedded devices.

Certificates and Private RSA Keys Statistics Many vendors include self-signed certificates inside their firmware images [136, 135]. Due to bad practices in both *release management* and *software design*, some vendors also include the private keys (e.g., PEM, GPG), as confirmed by recent advisories [144, 146].

We developed two simple plugins for our system which collect SSL certificates and private keys. These plugins also collect their fingerprints and check for empty or trivial passphrases. So far, we have been able to extract 109 private RSA keys from 428 firmware images and 56 self-signed SSL certificates out of 344 firmware images. In total, we obtained 41 self-signed SSL certificates together with their corresponding private RSA keys. By looking up those certificates in the public ZMap datasets [102], we were able to automatically locate about 35,000 active online devices.

For all these devices, if the certificate and private key are not regenerated on the first boot after a firmware update, HTTPS encryption can be easily decrypted by an attacker by simply downloading a copy of the firmware image. In addition, if both a regenerated and a firmware-shipped self-signed certificate are used interchangeably, the user of the device may still be vulnerable to man-in-the-middle (MITM) attacks.

Packaging Outdated and Vulnerable Software Another interesting finding relates to bad *release management* by embedded firmware vendors. Firmware images often rely on many third-party software and libraries. Those keep updating and have security fixes every now and then. OWASP Top Ten [205] lists *"Using Components with Known Vulnerabilities"* at position nine and underlines that *"upgrading to these new versions is critical"*.

In one particular case, we identified a relatively recently released firmware image that contained a kernel (version 2.4.20) that was built and packaged ten years after its initial release. In another case, we discovered that some recently released firmware images contained nine years old BusyBox versions.

Building Images as *root* While prototyping, putting together a build environment as fast as possible is very important. Unfortunately, sometimes the easiest solution is just to setup and run the entire toolchains as superuser.

Our analysis plugins extracted several compilation banners such as `Linux version 2.6.31.8-mv78100 (root@ubuntu) (gcc version 4.2.0 20070413 (prerelease)) Mon Nov 7 16:51:58 JST 2011` or `BusyBox v1.7.0 (2007-10-15 19:49:46 IST)`.

24% of the 450 unique banners we collected containing the `user@host` combinations were associated to the `root` user. In addition to this, among the 267 unique hostnames extracted from those banners, *ten* resolved to public IP addresses and *one* of these even accepted incoming SSH connections.

All these findings reveal a number of unsafe practices ranging from *build management* (e.g., build process done as `root`) to *infrastructure management* (e.g., build hosts reachable over public networks), to *release management* (e.g., usernames and hostnames not removed from production release builds).

Web Servers Configuration We developed a number of plugins to analyze the configuration files of web servers embedded in the firmware images such as `lighttpd.conf` or `boa.conf`. We then parsed the extracted files to retrieve specific configuration settings such as the running user, the documents root directory, and the file containing authentication secrets. We collected in total 847 distinct web server configuration files and the findings were discouraging. We found that in more than 81% of the cases the web servers were configured to run as a privileged user (i.e., having a setting such as `user=root`). This reveals unsafe practices of insecure design and configuration. Running the web server of an embedded device with unnecessarily high privileges can be extremely risky since the security of the entire device can be compromised by finding a vulnerability in one of the web components.

3.5 Case Studies

3.5.1 Backdoors in Plain Sight

Many backdoors in embedded systems have been reported recently, ranging from very simple cases [137] to others that were more difficult to discover [145, 224]. In one famous case [137], the backdoor was found to be activated by the string "`xmlset_roodkcableoj28840ybtide`" (i.e., "edit by 04882 joel backdoor" in reverse). This fully functional backdoor was affecting three vendors. Interestingly enough, this backdoor may have been detected earlier by a simple keyword matching on the open source release from the vendor[5].

Inspired by this case, we performed a string search in our dataset with various backdoor related keywords. Surprisingly, we found 1198 matches, in 326 firmware candidates.

Among those search results, several matched the firmware of a home automation device from a major vendor. According to download statistics from Google Play and Apple App Store, more than half a million users have downloaded an app for this device [17, 16].

We manually analyzed the firmware of this Linux-based embedded system and found that a daemon process listens on a network multicast address. This service allows execution of remote commands with `root` privileges without any authen-

tication to anybody in the local network. An attacker can easily gain full control if he can send multicast packets to the device.

We then used this example as a *seed* for our *correlation engine*. With this approach we found exactly the same backdoor in two other classes of devices from two different vendors. One of them was affecting 109 firmware images of 44 camera models of a major CCTV solutions vendor, *Vendor C*. The other case is affecting three firmware images for home routers of a major networking equipment vendor, *Vendor D*.

We investigated the issue and found that the affected devices were relying on the same provider of a System on a Chip (SoC) for networking devices. It seems that this backdoor is intended for system debugging, and is part of a development kit. Unfortunately we were not able to locate the source of this binary. We plan to acquire some of those devices to verify the exploitability of the backdoor.

3.5.2 Private SSL Keys

In addition to the backdoors left in firmware images from *Vendor C*, we also found many firmware images containing public and *private RSA key* pairs. Those unprotected keys are used to provide SSL access to the CCTV camera's web interface. Surprisingly, this private key is the same across many firmware images of the same brand.

Our platform automatically extracts the fingerprint of the public keys, private keys and SSL certificates. Those keys are then searched in ZMap's HTTPS survey database [102, 103]. *Vendor C*'s SSL certificate was found to be used by around 30K online IP addresses, most likely each corresponding to a single online device. We then fetched the web pages available at those addresses (without trying to authenticate). Surprisingly, we found CCTV cameras branded by another vendor – *Vendor B* – which appears to be an *integrator*. Upon inspection, cameras of *Vendor B* served *exactly the same* SSL certificate as cameras from *Vendor C* (including the SSL *Common Name*, and SSL *Organizational Unit* as well as many other fields of the SSL certificate). The only difference is that CCTV cameras of *Vendor B* returned branded authentication realms, error messages and logos. The *correlation engine* findings are summarized in Figure 3.4.

Unfortunately, the firmware images from *Vendor B* do not seem to be publicly available. We are planning to obtain a device to extract its firmware and to confirm our findings. We have reported these issues to the vendor. Nevertheless, it is very likely that devices from *Vendor B* are also vulnerable to the multicast packet backdoor given the clear relationship with *Vendor C* that that our platform discovered.

3.5.3 XSS in WiFi Enabled SD Cards?

SD cards are often more complex than one would imagine. Most SD cards actually contain a processor which runs firmware. This processor often manages functions such as the flash memory translation layer and wear leveling. Security issues have been previously shown on such SD cards [257].

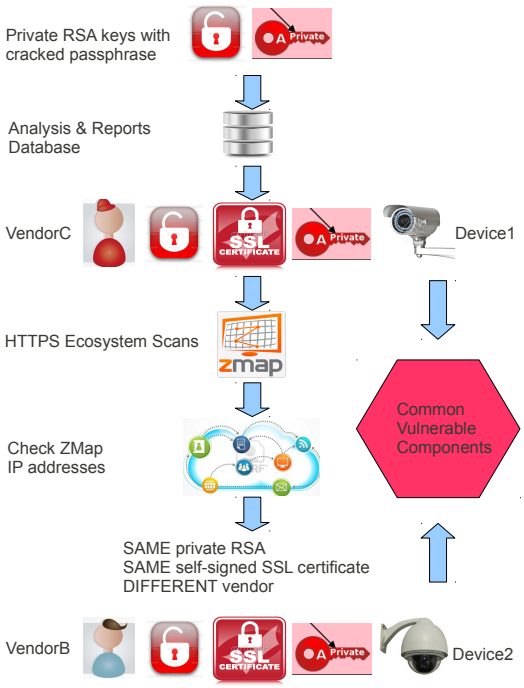


Fig. 3.4: Correlation engine and shared self-signed certificates clustering.

Some SD cards have an embedded WiFi interface with a full fledged web server. This interface allows direct access to the files on the SD card without ejecting it from the device in which it is inserted. It also allows administration of the SD card configuration (e.g., WiFi access points).

We manually found a Cross Site Scripting (XSS) vulnerability in one of these web interfaces, which consists of a `perl` based web application. As this web application does not have platform specific binary bindings, we were able to load the files inside a similar Boa web server on a PC and confirm the vulnerability.

Once we found the exact `perl` files responsible for the XSS, we used our correlation engine based on fuzzy hashes. With this we automatically found another SD card firmware that is vulnerable to the same XSS. Even though the `perl` files were slightly different, they were clearly identified as similar by the fuzzy hash. This correlation would not have been detected by a normal checksum or by a regular hash function.

The process is visualized in Figure 3.5. The file (*) was found vulnerable. Subsequently, we identified correlated files based on fuzzy hashing. Some of them were related to the same firmware or a previous version of the firmware of the *same* vendor (in red). Also, fuzzy hash correlation identified a similar file in a firmware from a *different* vendor (in orange) that is vulnerable to the same weakness. It further identified some non-vulnerable or non-related files from other vendors (in green).

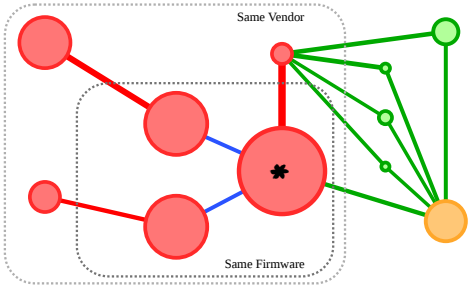


Fig. 3.5: Fuzzy hash clustering and vulnerability propagation. A vulnerability was propagated from a *seed file (*)* to other two files from the same firmware and three files from the same vendor (in red) as well as one file from another vendor (in orange). Also four non-vulnerable files (in green) have a strong correlation with vulnerable files. Edge thickness displays the strength of correlation between files.

Those findings are reported as `CVE-2013-5637` and `CVE-2013-5638`. We were also able to confirm this vulnerability and extend the list of affected versions for one of these vendors.

Such manual vulnerability confirmation does not scale. Hence, in the future we plan to integrate static analysis tools for web applications [91, 30, 150, 109, 2] in our process.

3.6 Ethical Discussion

Large-scale scans to test for the presence of vulnerabilities often raise serious ethical concerns. Even simple Internet-wide network scans may trigger alerts from intrusion detection systems (IDS) and may be perceived as an attack by the scanned networks.

In our study we were particularly careful to work within legal and ethical boundaries. First, we obtain firmware images either through user submission or through legitimate distribution mechanisms. In this case, our web crawler was designed to obey the `robots.txt` directives. Second, when we found new vulnerabilities we worked together with vendors and CERTs to confirm the devices vulnerabilities and to perform responsible disclosure. Finally, the license of some firmware images may not allow redistribution. Therefore, the public web submission interface limits the ability to access firmware contents only to the users who uploaded the corresponding firmware image. Other users can only access anonymized reports. We are currently investigating ways to make the full dataset available for research purposes to well identified research institutions.

3.7 Related Work

Several studies have been proposed to asses the security of embedded devices by scanning the Internet. For instance, Cui et al. [86, 87] present a wide-scale Internet scan to first recognize devices that are known to be shipped with default password, and then to confirm that these devices are indeed still vulnerable by attempting to login into them. Heninger et al. [139] performed the largest ever network survey of TLS and SSH servers, showing that vulnerable keys are surprisingly widespread and that the vast majority appear to belong to headless or embedded devices. ZMap [103] is an efficient and fast network scanner, that allows to scan the complete Internet IPv4 address space in less than one hour. While the scans are not especially targeted to embedded devices, in our work we reuse the SSL certificates scans performed by ZMap [102]. Similar scans were targeting specific vulnerabilities often present in embedded devices [134, 7]. Such wide-scale scans are mainly targeted at discovering online devices affected by already known vulnerabilities, but in some cases they can help to discover new flaws. However, many categories of flaws cannot be discovered by such scans. Some online services like Shodan [9] provide a global updated view on publicly available devices and web services. This easy-to-use research tool allows security researchers to identify systems worldwide that are potentially exposed or exploitable.

Unpacking firmware images is a known problem, and several tools for this purpose exist. Binwalk [3] is a firmware analysis toolbox that provides various methods and tools for extraction, inspection and reverse engineering of firmware images or other binary blobs. FRAK [83] is a framework to unpack, analyze, and repack firmware images of embedded devices. `FRAK` was never publicly released and reportedly supports only a few firmware formats (e.g., Cisco IP phones and IOS, HP laser printers). The Binary Analysis Toolkit (BAT) [138, 241] was originally designed to detect GPL license violations, mainly by comparing strings in a firmware image to strings present in open source software distributions. For this purpose `BAT` has to unpack firmware images. Unfortunately, as we show in Section 3.3, none of these tools are accurate and complete enough to be used *as is* in our framework.

There are many examples of security analysis of embedded systems [263]. Several network card firmware images have been analyzed and modified to insert a backdoor [96, 100] or to extend their functionality [42]. Davidson et al. [93] propose `FIE`, built on top of the `KLEE` symbolic execution engine, to incorporate new symbolic execution techniques. It can be used to verify security properties of some simple firmware images often found in practice. Zaddach et al. [262] describe Avatar, a dynamic analysis platform for firmware security testing. In Avatar, the instructions are executed in an emulator, while the IO accesses to the embedded system's peripherals are forwarded to the real device. This allows a security engineer to apply a wide range of advanced dynamic analysis techniques like tracing, tainting and symbolic execution.

A large set of firmware images of Xerox devices were reverse-engineered by Costin [78] leading to the discovery of hidden PostScript commands. Such commands allow an attacker to e.g., dump a device's memory, recover passwords, passively scan the network and more generically interact with devices' OS layers. Such attacks could be delivered to printers via web pages, applets, MS Word and other standard printed documents [77].

Bojinov et al. [46] conducted an assessment of the security of current embedded management interfaces. The study, conducted on real physical devices, found vulnerabilities in 21 devices from 16 different brands, including network switches, cameras, photo frames, and lights-out management modules. Along with these, a new class of vulnerabilities was discovered, namely *cross-channel scripting (XCS)* [45]. While XCS vulnerabilities are not particular to embedded devices, embedded devices are probably the most affected population. In a similar study, the authors manually analyzed ten Small Office/Home Office (SOHO) routers [143] and discovered at least two vulnerabilities per device.

Looking at insecure (remote) firmware updates, researchers reported the possibility to arbitrarily inject malware into the firmware of a printer [78, 85]. Chen [65] and Miller [187] presented techniques and implications of exploiting Apple firmware updates. In a similar direction, Basnight et al. [33] examined the vulnerability of PLCs to intentional firmware modifications. A general firmware analysis methodology is presented, and an experiment demonstrates how legitimate firmware can be updated on an Allen-Bradley ControlLogix L61 PLC. Zaddach et al. [265] explore the consequences of a backdoor injection into the firmware of a hard disk drive and uses it to exfiltrate data.

French and Casey [34] present fuzzy hashing techniques in applied malware analysis. Authors used `ssdeep` on *CERT Artifact Catalog* database containing $10.7M$ files. The study underlines the two fundamental challenges to operational usage of fuzzy hashing at scale: timeliness of results, and usefulness of results. To reduce the quadratic complexity of the comparisons, they propose assigning files into "bins" based on the block and file sizes. This approach, for their particular dataset and bins partitioning strategy, allowed for a given fuzzy hash to reduce the search space on average by 83.1%.

Finally, Bailey et al. [29] and Bayer et al. [35] propose efficient clustering approaches to identify and group malware samples at large scale. Authors perform dynamic analysis to obtain the execution traces of malware programs or obtain a description of malware behavior in terms of system state changes. These are then generalized into behavioral profiles which serve as input to an efficient clustering algorithm that allows authors to handle sample sets that are an order of magnitude larger than previous approaches. Unfortunately, this approach cannot be applied in our framework since dynamic analysis is unfeasible due to the heterogeneity of architectures used in firmware images.

3.8 Conclusion

In this paper we conducted a large-scale static analysis of embedded firmwares. We showed that a broader view on firmware is not only beneficial, but actually necessary for discovering and analyzing vulnerabilities of embedded devices. Our study helps researchers and security analysts to put the security of particular devices in context, and allows them to see how known vulnerabilities that occur in one firmware reappear in the firmware of other manufacturers.

We plan to continue collecting new data and extend our analysis to all the firmware images we downloaded so far. Moreover, we want to extend our system with more sophisticated static analysis techniques that allow a more in-depth study of each firmware image. This approach shows a lot of potential and besides the few previously mentioned case studies it can lead to new interesting results such as the ones recently found by Costin et al. [79].

The summarized datasets are available at:
http://firmware.re/usenixsec14.

4

Dowsing for overflows: A guided fuzzer to find buffer boundary violations

Preamble: Relation to the Research Roadmap

The Red Book, in Chapter 4, defines a vulnerability as a weakness or flaw in one or more software components that can be exploited to compromise the integrity, confidentiality, or availability of a system and its information resources. Besides software, vulnerabilities may exist in other aspects of a system, including protocol design, hardware, system configuration, and operational procedures. Despite significant advances in software protection and attack mitigation techniques, exploitable vulnerabilities are continuously being discovered even in the latest versions of widely used applications, programming libraries, operating systems, online services, embedded software, and other programs.

Despite many years of security research and engineering, software vulnerabilities remain one of the primary culprits that reduce a systems information assurance. Conversely, finding vulnerabilities before the software is released increases the system's security.

One of the four main classes of vulnerability defined in the Red Book is known as memory corruption, of which the buffer overflow is the most prominent example. In this chapter, we will explore new techniques to find such vulnerabilities in an automated fashion. Many of our legacy systems, even in critical infrastructures (Chapter 6 and 9 in the Red Book) are written in unsafe programming languages like C and C++, and thus vulnerable. Memory errors such as buffer overflows are explicitly identified as ranking among the most dangerous software errors.

Istvan Haller and Asia Slowinska, Matthias Neugschwandtner, Herbert Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations" 22nd USENIX Security Symposium (USENIX Security 13), Washington, D.C., USA, August 2013

Abstract *Dowser* is a 'guided' fuzzer that combines taint tracking, program analysis and symbolic execution to find buffer overflow and underflow vulnerabilities buried deep in a program's logic. The key idea is that analysis of a program lets us pinpoint the right areas in the program code to probe and the appropriate inputs to do so.

Intuitively, for typical buffer overflows, we need consider only the code that accesses an array in a loop, rather than all possible instructions in the program. After finding all such candidate sets of instructions, we rank them according to an estimation of how likely they are to contain interesting vulnerabilities. We then subject the most promising sets to further testing. Specifically, we first use taint analysis to determine which input bytes influence the array index and then execute the program symbolically, making only this set of inputs symbolic. By constantly steering the symbolic execution along branch outcomes most likely to lead to overflows, we were able to detect deep bugs in real programs (like the `nginx` webserver, the `inspircd` IRC server, and the `ffmpeg` videoplayer). Two of the bugs we found were previously undocumented buffer overflows in `ffmpeg` and the `poppler` PDF rendering library.

4.1 Introduction

We discuss *Dowser*, a 'guided' fuzzer that combines taint tracking, program analysis and symbolic execution, to find buffer overflow bugs buried deep in the program's logic.

Buffer overflows are perennially in the top 3 most dangerous software errors [90] and recent studies suggest this will not change any time soon [247, 230]. There are two ways to handle them. Either we harden the software with memory protectors that terminate the program when an overflow occurs (at runtime), or we track down the vulnerabilities before releasing the software (e.g., in the testing phase).

Memory protectors include common solutions like shadow stacks and canaries [80], and more elaborate compiler extensions like WIT [22]. They are effective in preventing programs from being exploited, but they do not remove the overflow bugs themselves. Although it is better to crash than to allow exploitation, crashes are undesirable too!

Thus, vendors prefer to squash bugs beforehand and typically try to find as many as they can by means of fuzz testing. Fuzzers feed programs invalid, unexpected, or random data to see if they crash or exhibit unexpected behavior[1]. As an example, Microsoft made fuzzing mandatory for every untrusted interface for every product, and their fuzzing solution has been running 24/7 since 2008 for a total of over 400 machine years [124].

Unfortunately, the effectiveness of most fuzzers is poor and the results rarely extend beyond shallow bugs. Most fuzzers take a 'blackbox' approach that focuses

[1] See `http://www.fuzzing.org/` for a collection of available fuzzers

on the input format and ignores the tested software target. Blackbox fuzzing is popular and fast, but misses many relevant code paths and thus many bugs. Blackbox fuzzing is a bit like shooting in the dark: you have to be lucky to hit anything interesting.

Whitebox fuzzing, as implemented in [124, 54, 71], is more principled. By means of symbolic execution, it exercises all possible execution paths through the program and thus uncovers all possible bugs – although it may take years to do. Since full symbolic execution is slow and does not scale to large programs, it is hard to use it to find complex bugs in large programs [54, 71]. In practice, the aim is therefore to first cover as much unique code as possible. As a result, bugs that require a program to execute the same code many times (like buffer overflows) are hard to trigger except in very simple cases.

Eventual completeness, as provided by symbolic execution, is both a strength and a weakness, and in this paper, we evaluate the exact opposite strategy. Rather than testing all possible execution paths, we perform *spot checks* on a small number of code areas that look likely candidates for buffer overflow bugs and test each in turn.

The drawback of our approach is that we execute a symbolic run for each candidate code area—in an iterative fashion. Moreover, we can discover buffer overflows only in the loops that we can exercise. On the other hand, by homing in on promising code areas directly, we speed up the search considerably, and manage to find complicated bugs in real programs that would be hard to find with most existing fuzzers.

Contributions The goal we set ourselves was to develop an efficient fuzzer that actively *searches* for buffer overflows *directly*. The key insight is that careful analysis of a program lets us pinpoint the right places to probe and the appropriate inputs to do so. The main contribution is that our fuzzer directly zooms in on these buffer overflow candidates and explores a novel 'spot-check' approach in symbolic execution.

To make the approach work, we need to address two main challenges. The first challenge is *where* to steer the execution of a program to increase the chances of finding a vulnerability. Whitebox fuzzers 'blindly' try to execute as much of the program as possible, in the hope of hitting a bug eventually. Instead, *Dowser* uses information about the target program to identify code that is most likely to be vulnerable to a buffer overflow.

For instance, buffer overflows occur (mostly) in code that accesses an array in a loop. Thus, we look for such code and ignore most of the remaining instructions in the program. Furthermore, *Dowser* performs static analysis of the program to *rank* such accesses. We will evaluate different ranking functions, but the best one so far ranks the array accesses according to complexity. The intuition is that code with convoluted pointer arithmetic and/or complex control flow is more prone to memory errors than straightforward array accesses. Moreover, by focusing on such

code, *Dowser* prioritizes bugs that are complicated—typically, the kind of vulnerabilities that static analysis or random fuzzing cannot find. The aim is to reduce the time wasted on shallow bugs that could also have been found using existing methods. Still, other rankings are possible also, and *Dowser* is entirely agnostic to the ranking function used.

The second challenge we address is *how* to steer the execution of a program to these "interesting" code areas. As a baseline, we use *concolic* execution [256]: a combination of concrete and symbolic execution, where the concrete (fixed) input starts off the symbolic execution. In *Dowser*, we enhance concolic execution with two optimizations.

First, we propose a new path selection algorithm. As we saw earlier, traditional symbolic execution aims at code coverage—maximizing the fraction of individual branches executed [54, 124]. In contrast, we aim for *pointer value* coverage of *selected* code fragments. When *Dowser* examines an interesting pointer dereference, it steers the symbolic execution along branches that are likely to alter the value of the pointer.

Second, we reduce the amount of symbolic input as much as we can. Specifically, *Dowser* uses dynamic taint analysis to determine which input bytes influence the pointers used for array accesses. Later, it treats only these inputs as symbolic. While taint analysis itself is not new, we introduce novel optimizations to arrive at a set of symbolic inputs that is as *accurate* as possible (with neither too few, nor too many symbolic bytes).

In summary, *Dowser* is a new fuzzer targeted at vendors who want to test their code for buffer overflows and underflows. We implemented the analyses of *Dowser* as LLVM [169] passes, while the symbolic execution step employs S2E [71]. Finally, *Dowser* is a *practical* solution. Rather than aiming for all possible security bugs, it specifically targets the class of buffer overflows (one of the most, if not the most, important class of attack vectors for code injection). So far, *Dowser* found several real bugs in complex programs like `nginx`, `ffmpeg`, and `inspircd`. Most of them are extremely difficult to find with existing symbolic execution tools.

Assumptions and outline Throughout this paper, we assume that we have a test suite that allows us to reach the array accesses. Instructions that we cannot reach, we cannot test. In the remainder, we start with a big picture and the running example (Section 4.2). Then, we discuss the three main components of *Dowser* in turn: the selection of interesting code fragments (Section 4.3), the use of dynamic taint analysis to determine which inputs influence the candidate instructions (Section 4.4), and our approach to nudge the program to trigger a bug during symbolic execution (Section 4.5). We evaluate the system in Section 4.6, discuss the related projects in Section 4.7. We conclude in Section 4.8.

4.2 Big picture

The main goal of *Dowser* is to manipulate the pointers that instructions use to access an array in a loop, in the hope of forcing a buffer overrun or underrun.

4.2.1 Running example

Throughout the paper, we will use the function in Figure 4.1 to illustrate how *Dowser* works. The example is a simplified version of a buffer underrun vulnerability in the `nginx-0.6.32` web server [14]. A specially crafted input tricks the program into setting the `u` pointer to a location outside its buffer boundaries. When this pointer is later used to access memory, it allows attackers to overwrite a function pointer, and execute arbitrary programs on the system.

Figure 4.1 presents only an excerpt from the original function, which in reality spans approximately 400 lines of C code. It contains a number of additional options in the `switch` statement, and a few nested conditional `if` statements. This complexity severely impedes detecting the bug by both static analysis tools and symbolic execution engines. For instance, when we steered S2E [71] all the way down to the vulnerable function, and made solely the seven byte long uri path of the HTTP message symbolic, it took over 60 minutes to track down the problematic scenario. A more scalable solution is necessary in practice. Without these hints, S2E did not find the bug at all during an eight hour long execution.[2] In contrast, *Dowser* finds it in less than 5 minutes.

The primary reason for the high cost of the analysis in S2E is the large number of conditional branches which depend on (symbolic) input. For each of the branches, symbolic execution first checks whether either the condition or its negation is satisfiable. When both branches are feasible, the default behavior is to examine both. This procedure results in an exponentially growing number of paths.

This real world example shows the need for (1) focusing the powerful yet expensive symbolic execution on the most interesting cases, (2) making informed branch choices, and (3) minimizing the amount of symbolic data.

4.2.2 High-level overview

Figure 4.2 illustrates the overall *Dowser* architecture.

First, it performs a data flow analysis of the target program, and ranks all instructions that access buffers in loops ①. While we can rank them in different ways and *Dowser* is agnostic as to the ranking function we use, our experience so far is that an estimation of complexity works best. Specifically, we rank calculations and conditions that are more complex higher than simple ones. In Figure 4.1, `u` is involved in three different operations, i.e., `u++`, `u--`, and `u-=4`, in multiple instructions inside a loop. As we shall see, these intricate computations place the dereferences of `u` in the top 3% of the most complex pointer accesses across `nginx`.

[2] All measurements in the paper use the same environment as in Section 4.6.

In the second step ②, *Dowser* repeatedly picks high-ranking accesses, and selects test inputs which exercise them. Then, it uses dynamic taint analysis to determine which input bytes influence pointers dereferenced in the candidate instructions. The idea is that, given the format of the input, *Dowser* fuzzes (i.e., treats as symbolic), only those fields that affect the potentially vulnerable memory accesses, and keeps the remaining ones unchanged. In Figure 4.1, we learn that it is sufficient to treat the uri path in the HTTP request as symbolic. Indeed, the computations inside the vulnerable function are independent of the remaining part of the input message.

Next ③, for each candidate instruction and the input bytes involved in calculating the array pointer, *Dowser* uses symbolic execution to try to nudge the program toward overflowing the buffer. Specifically, we execute symbolically the loop that contains the candidate instructions (and thus should be tested for buffer overflows)—treating only the relevant bytes as symbolic. As we shall see, a new path selection algorithm helps to guide execution to a possible overflow quickly.

Finally, we detect any overflow that may occur. Just like in whitebox fuzzers, we can use any technique to do so (e.g., Purify, Valgrind [198], or BinArmor [228]). In our work, we use Google's AddressSanitizer [220] ④. It instruments the protected program to ensure that memory access instructions never read or write so called, "poisoned" red zones. Red zones are small regions of memory inserted inbetween any two stack, heap or global objects. Since they should never be addressed by the program, an access to them indicates an illegal behavior. This policy detects sequential buffer over- and underflows, and some of the more sophisticated pointer corruption bugs. This technique is beneficial when searching for new bugs since it will also trigger on silent failures, not just application crashes. In the case of `nginx`, AddressSanitizer detects the underflow when the `u` pointer reads memory outside its buffer boundaries (line 33).

We explain step ① (static analysis) in Section 4.3, step ② (taint analysis) in Section 4.4, and step ③ (guided execution) in Section 4.5.

4.3 Dowsing for candidate instructions

Previous research has shown that software complexity metrics collected from software artifacts are helpful in finding vulnerable code components [120, 266, 221, 200]. However, even though complexity metrics serve as useful indicators, they also suffer from low precision or recall values. Moreover, most of the current approaches operate at the granularity of modules or files, which is too coarse for the directed symbolic execution in *Dowser*.

As observed by Zimmermann et al. [266], we need metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns. In principle, *Dowser* can work with any metric capable of ranking groups of instructions that access buffers in a loop. So, the question is how to design a good metric for complexity that satisfies this criterion? In the remainder of this section,

we introduce one such metric: a heuristics-based approach that we specifically designed for the detection of potential buffer overflow vulnerabilities.

We leverage a primary pragmatic reason behind complex buffer overflows: convoluted pointer computations are hard to follow by a programmer. Thus, we focus on 'complex' array accesses realized inside loops. Further, we limit the analysis to pointers which evolve together with loop induction variables, i.e., are repeatedly updated to access (various) elements of an array.

Using this metric, *Dowser* ranks buffer accesses by evaluating the complexity of data- and control-flows involved with the array index (pointer) calculations. For each loop in the program, it first statically determines (1) the set of all instructions involved in modifying an array pointer (we will call this a pointer's *analysis group*), and (2) the conditions that guard this analysis group, e.g., the condition of an `if` or `while` statement containing the array index calculations. Next, it labels all such sets with scores reflecting their complexity. We explain these steps in detail in Sections 4.3.1, 4.3.2, and 4.3.3.

4.3.1 Building analysis groups

Suppose a pointer `p` is involved in an "interesting" array access instruction $\texttt{acc}_\texttt{p}$ in a loop. The *analysis group* associated with $\texttt{acc}_\texttt{p}$, $\texttt{AG(acc}_\texttt{p}\texttt{)}$, collects all instructions that influence the value of the dereferenced pointer during the execution of the loop.

To determine $\texttt{AG(acc}_\texttt{p}\texttt{)}$, we compute an intraprocedural data flow graph representing operations in the loop that compute the value of `p` dereferenced in $\texttt{acc}_\texttt{p}$. Then, we check if the graph contains cycles. A cycle indicates that the value of `p` in a previous loop iteration affects its value in the current one, so `p` depends on the loop induction variable.

As mentioned before, this part of our work is built on top of the LLVM [169] compiler infrastructure. The static single assignment (SSA) form provided by LLVM translates directly to data flow graphs. Figure 4.3 shows an example. Observe that, since all dereferences of pointer `u` share their data flow graph, they also form a single analysis group. Thus, when *Dowser* later tries to find an illegal array access within this analysis group, it tests all the dereferences at the same time—there is no need to consider them separately.

4.3.2 Conditions guarding analysis groups

It may happen that the data flow associated with an array pointer is simple, but the value of the pointer is hard to follow due to some complex control changes. For this reason, *Dowser* ranks also control flows: the conditions that influence an analysis group.

Say that an instruction manipulating the array pointer `p` is guarded by a condition on a variable `var`, e.g., `if(var<10){*p++=0;}`. If the value of `var` is difficult to keep track of, so is the value of `p`. To assess the complexity of `var`, *Dowser* analyzes its data flow, and determines the analysis group, `AG(var)` (as

discussed in Section 4.3.1). Moreover, we recursively analyze the analysis groups of other variables influencing `var` and `p` inside the loop. Thus, we obtain a number of analysis groups which we rank in the next step (Section 4.3.3).

4.3.3 Scoring array accesses

For each array access realized in a loop, *Dowser* assesses the complexity of the analysis groups constructed in Sections 4.3.1 and 4.3.2. For each analysis group, it considers all instructions, and assigns them points. The more points an AG cumulatively scores, the more complex it is. The overall rank of the array access is determined by the maximum of the scores. Intuitively, it reflects the most complex component.

The scoring algorithm should provide roughly the same results for semantically identical code. For this reason, we enforce the optimizations present in the LLVM compiler (e.g., to eliminate common subexpressions). This way, we minimize the differences in (the amount of) instructions arising from the compiler options. Moreover, we analyzed the LLVM code generation strategies, and defined a powerful set of *equivalence rules*, which minimize the variation in the scores assigned to syntactically different but semantically equivalent code. We highlight them below.

Table 4.1 introduces all types of instructions, and discusses their impact on the final score. In principle, all common instructions involved in array index calculations are of the order of 10 points, except for the two instructions that we consider risky: pointer casts and functions that return non-pointer values used in pointer calculation.

A buffer underrun vulnerability in `nginx`

```
[1] int ngx_http_parse_complex_uri(ngx_http_request_t *r)
[2] {
[3]     state = sw_usual;
[4]     u_char* p = r->uri_start;   // user input
[5]     u_char* u = r->uri.data; // store normalized uri here
[6]     u_char ch = *p++;           // the current character
[7]
[8]     while (p <= r->uri_end) {
[9]         switch (state) {
[10]            case sw_usual:
[11]                if (ch == '/')
[12]                    state = sw_slash; *u++ = ch;
[13]                else /* many more options here */
[14]                ch = *p++; break;
[15]
[16]            case sw_slash:
[17]                if (ch == '/')
[18]                    *u++ = ch;
[19]                else if (ch == '.')
[20]                    state = sw_dot; *u++ = ch;
[21]                else /* many more options here */
[22]                ch = *p++; break;
[23]
[24]            case sw_dot:
[25]                if (ch == '.')
[26]                    state = sw_dot_dot; *u++ = ch;
[27]                else /* many more options here */
[28]                ch = *p++; break;
[29]
[30]            case sw_dot_dot:
[31]                if (ch == '/')
[32]                    state = sw_slash; u -=4;
[33]                    while (*(u-1) != '/') u--;
[34]                else /* many more options here */
[35]                ch = *p++; break;
[36]        }
[37]    }
[38] }
```

`Nginx` is a web server—in terms of market share across the million busiest sites, it ranks third in the world. At the time of writing, it hosts about 22 million domains worldwide. Versions prior to 0.6.38 had a particularly nasty vulnerability [14].

When `nginx` receives an HTTP request, the parsing function `nginx_http_parse_complex_uri`, first normalizes a uri path in `p=r->uri_start` (line 4), storing the result in a heap buffer pointed to by `u=r->uri.data` (line 5). The `while-switch` implements a state machine that consumes the input one character at a time, and transform it into a canonical form in `u`.

The source of the vulnerability is in the `sw_dot_dot` state. When provided with a carefully crafted path, `nginx` wrongly sets the beginning of `u` to a location somewhere below `r->uri.data`. Suppose the uri is `"//../foo"`. When `p` reaches `"/foo"`, `u` points to `(r->uri.data+4)`, and `state` is `sw_dot_dot` (line 30). The routine now decreases `u` by 4 (line 32), so that it points to `r->uri.data`. As long as the memory below `r->uri.data` does not contain the character `"/"`, `u` is further decreased (line 33), even though it crosses buffer boundaries. Finally, the user provided input (`"foo"`) is copied to the location pointed to by `u`.

In this case, the overwritten buffer contains a pointer to a function, which will be eventually called by `nginx`. Thus the vulnerability allows attackers to modify a function pointer, and execute an arbitrary program on the system.

It is a complex bug that is hard to find with existing solutions. The many conditional statements that depend on symbolic input are problematic for symbolic execution, while input-dependent indirect jumps are also a bad match for static analysis.

Fig. 4.1: A simplified version of a buffer underrun vulnerability in `nginx`.

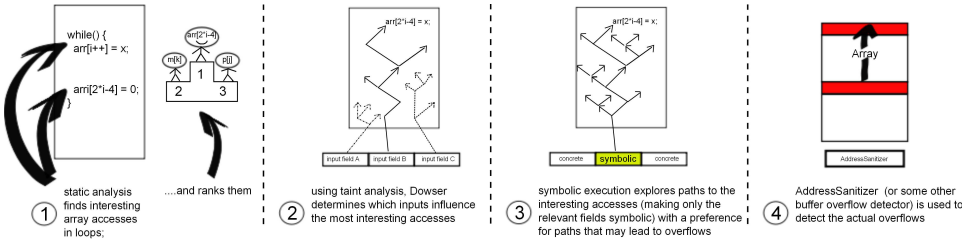


Fig. 4.2: *Dowser*– high-level overview.

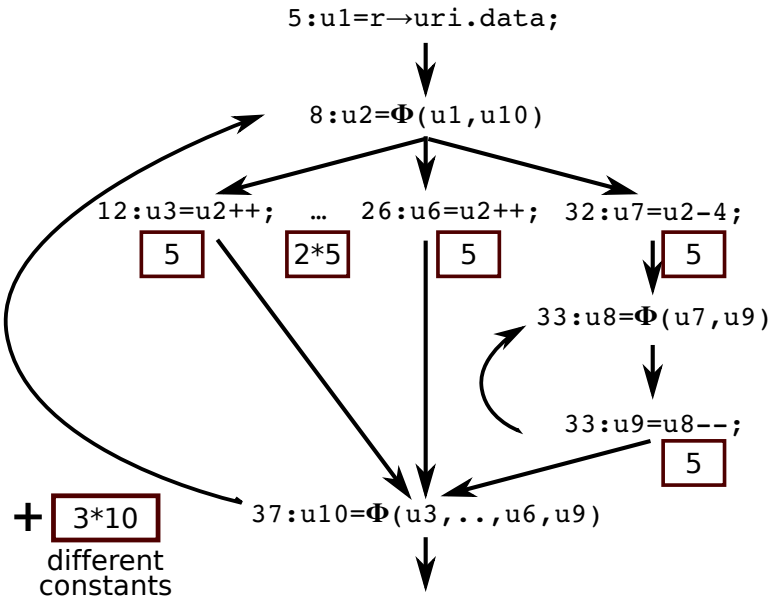


Fig. 4.3: Data flow graph and analysis group associated with the pointer `u` from Figure 4.1. For the sake of clarity, the figure presents pointer arithmetic instructions in pseudo code. The PHI nodes represent locations where data is merged from different control-flows. The numbers in the boxes represent points assigned by *Dowser*.

Instructions	Rationale/Equivalence rules	Points
Array index manipulations		
Basic index arithmetic instr., i.e., addition and subtraction	`GetElemPtr`, that increases or decreases a pointer by an index, scores the same. Thus, operations on pointers are equivalent to operations on offsets. An instruction scores 1 if it modifies a value which is not passed to the next loop iteration.	1 or 5
Other index arithmetic instr. e.g., division, shift, or xor	These instructions involve more complex pointer calculations than the standard `add` or `sub`. Thus, we penalize them more.	10
Different constant values	Multiple constants used to modify a pointer make its value hard to follow. It is easier to keep track of a pointer that always increases by the same value.	10 per value
Constants used to access fields of structures	We assume that compilers handle accesses to structures correctly. We only consider constants used to compute the index of an array, and not the address of a field.	0
Numerical values determined outside the loop	Though in the context of the loop they are just constants, the compiler cannot predict their values. Thus they are difficult to reason about and more error prone.	30
Non-inlined functions returning non-pointer values	Since decoupling the computation of a pointer from its use might easily lead to mistakes, we heavily penalize this operation.	500
Data movement instructions	Moving (scalar or pointer) data does not add to the complexity of computations.	0
Pointer manipulations		
Load a pointer calculated outside the loop	It denotes retrieving the base pointer of an object, or using memory allocators. We treat all *remote* pointers in the same way - all score 0.	0
`GetElemPtr`	An LLVM instruction that computes a pointer from a base and offset(s). (See `add`.)	1 or 5
Pointer cast operations	Since the casting instructions often indicate operations that are not equivalent to the standard pointer manipulations (listed above), they are worth a close inspection.	100

Table 4.1: Overview of the instructions involved in pointer arithmetic operations, and their penalty points.

The absolute penalty for each type of instruction is not very important. However, we ensure that the points reflect the difference in complexity between various code fragments, instead of giving all array accesses the same score. That is, instructions that complicate the array index contribute to the score, and instructions that complicate the index a lot also score very high, relative to other instructions. In Section 4.6, we compare our complexity ranking to alternatives.

4.4 Using tainting to find inputs that matter

Once *Dowser* has ranked array accesses in loops in order of complexity, we examine them in turn. Typically, only a small segment of the input affects the execution of a particular analysis group, so we want to search for a bug by modifying solely this part of the input, while keeping the rest constant (refer to Section 4.5). In the current section, we explain how *Dowser* identifies the link between the components of the program input and the different analysis groups. Observe that this result also benefits other bug finding tools based on fuzzing, not just *Dowser* and concolic execution.

We focus our discussion on an analysis group $\texttt{AG}(\texttt{acc}_\texttt{p})$ associated with an array pointer dereference $\texttt{acc}_\texttt{p}$. We assume that we can obtain a test input `I` that exercises the potentially vulnerable analysis group. While this may not always be true, we believe it is a reasonable assumption. Most vendors have test suites to test their software and they often contain at least one input which exercises each *complex* loop.

4.4.1 Baseline: dynamic taint analysis

As a basic approach, *Dowser* performs dynamic taint analysis (DTA) [199] on the input `I` (tainting each input byte with a unique color, and propagating the colors on data movement and arithmetic operations). Then, it logs all colors and input bytes involved in the instructions in $\texttt{AG}(\texttt{acc}_\texttt{p})$. Given the format of the input, *Dowser* maps these bytes to individual fields. In Figure 4.1, *Dowser* finds out that it is sufficient to treat `uri` as symbolic.

The problem with DTA, as sketched above, is that it misses *implicit flows* (also called *control dependencies*) entirely [113, 155]. Such flows have no direct assignment of a tainted value to a variable—which would be propagated by DTA. Instead, the value of a variable is completely determined by the value of a tainted variable in a condition. In Figure 4.1, even though the value of `u` in line 12 is dependent on the tainted character `ch` in line 11, the taint does not flow directly to `u`, so DTA would not report the dependency. Implicit flows are notoriously hard to track [226, 61], but ignoring them completely reduces our accuracy. *Dowser* therefore employs a solution that builds on the work by Bao et al. [32], but with a novel optimization to increase the accuracy of the analysis (Section 4.4.2).

Like Bao et al. [32], *Dowser* implements *strict control dependencies*. Intuitively, we propagate colors only on the most informative (or, information preserv-

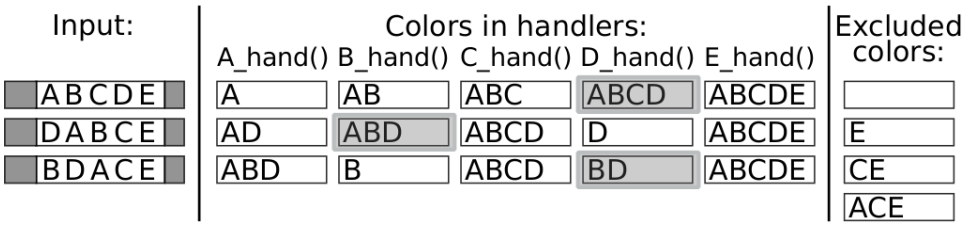


Fig. 4.4: The figure shows how *Dowser* shuffles an input to determine which fields really influence an analysis group. Suppose a parser extracts fields of the input one by one, and the analysis group depends on the fields B and D (with colors B and D, respectively). *Colors in handlers* show on which fields the subsequent handlers are strictly dependent [32], and the shaded rectangle indicates the colors propagated to the analysis group. *Excluded colors* are left out of our analysis.

ing) dependencies. Specifically, we require a direct comparison between a tainted variable and a compile time constant. For example, in Figure 4.1, we propagate the color of `ch` in line 11 to the variables `state` and `u` in line 12. However, we would keep `state` and `u` untainted if the condition in line 11 for instance had been either `"if(ch!='/')"` or `"if(ch<'/')"`. As implicit flows are not the focus of this paper we refer interested readers to [32] for details.

4.4.2 Field shifting to weed out false dependencies

Improving on the handling of strict control dependencies by Bao et al. [32], described above, *Dowser* adds a novel technique to prevent overtainting due to false dependencies. The problems arise when the order of fields in an input format is not fixed, e.g., as in HTTP, SMTP (and the commandline for most programs). The approach from [32] may falsely suggest that a field is dependent on all fields that were extracted so far.

For instance, `lighttpd` reads new header fields in a loop and compares them to various options, roughly as follows:

```
while () {
  if(cmp(field, "Content") == 0)
  ...
  else if(cmp(field, "Range") == 0)
  ...
  else exit (-1);
  field = extract_new_header_field();
}
```

As the parser tests for equivalence, the implicit flow will propagate from one field to the next one, even if there is no real dependency at all! Eventually, the last field appears to depend on the whole header.

Dowser determines which options really matter for the instructions in an analysis group by *shifting* the fields whose order is not fixed. Refer to Figure 4.4, and

suppose we have run the program with options A, B, C, D, and E, and our analysis group really depends on B and D. Once the message gets processed, we see that the AG does not depend on E, so E can be excluded from further analysis. Since the last observed color, D, has a direct influence on the AG, it is a true dependence. By performing a circular shift and re-trying with the order D, A, B, C, E, *Dowser* finds only the colors corresponding to A, B, D. Thus, we can leave C out of our analysis. After the next circular shift, *Dowser* reduces the colors to B and D only.

The optimization is based on two observations: (1) the last field propagated to the AG has a direct influence on the AG, so it needs to be kept, (2) all fields beyond this one are guaranteed to have no impact on the AG. By performing circular shifts, and running DTA on the updated input, *Dowser* drops the undue dependencies.

Even though this optimization requires some minimal knowledge of the input, we do not need full understanding of the input grammar, like the contents or effects of fields. It is sufficient to identify the fields whose order is not fixed. Fortunately, such information is available for many applications—especially when vendors test their own code.

4.5 Exploring candidate instructions

Once we have learnt which part of the program input influences the analysis group $\texttt{AG(acc}_{\texttt{p}}\texttt{)}$, we fuzz this part, and we try to nudge the program toward using the pointer `p` in an illegal way. More technically, we treat the interesting component of the input as symbolic, the remaining part as fixed (concrete), and we execute the loop associated with $\texttt{AG(acc}_{\texttt{p}}\texttt{)}$ symbolically.

However, since in principle the cost of a complete loop traversal is exponential, loops present one of the hardest problems for symbolic execution [125]. Therefore, when analyzing a loop, we try to select those paths that are most promising in our context. Specifically, *Dowser* prioritizes paths that show a potential for knotty pointer arithmetic. As we show in Section 4.6, our technique significantly optimizes the search for an overflow.

Dowser's loop exploration procedure has two main phases: learning, and bug finding. In the *learning phase*, *Dowser* assigns each branch in the loop a weight approximating the probability that a path following this direction contains new pointer dereferences. The weights are based on statistics on the variety of pointer values observed during an execution of a short symbolic input.

Next, in the *bug finding phase*, *Dowser* uses the weights determined in the first step to filter our uninteresting parts of the loop, and prioritize the important paths. Whenever the weight associated with a certain branch is 0, *Dowser* does not even try to explore it further. In the vulnerable `nginx` parsing loop from which Figure 4.1 shows an excerpt, only 19 out of 60 branches scored a non-zero value, so were considered for the execution. In this phase, the symbolic input represents a real world scenario, so it is relatively long. Therefore, it would be prohibitively expensive to be analyzed using a popular symbolic execution tool.

In Section 4.5.1, we briefly review the general concept of concolic execution, and then we discuss the two phases in Sections 4.5.2 and 4.5.3, respectively.

4.5.1 Baseline: concrete + symbolic execution

Like DART and SAGE [121, 124], *Dowser* generates new test inputs by combining concrete and symbolic execution. This technique is known as *concolic* execution [219]. It runs the program on a concrete input, while gathering symbolic constraints from conditional statements encountered along the way. To test alternative paths, it systematically negates the collected constraints, and checks whether the new set is satisfiable. If so, it yields a new input. To bootstrap the procedure, *Dowser* takes a test input which exercises the analysis group $\texttt{AG(acc}_\texttt{p}\texttt{)}$.

As mentioned already, a challenge in applying this approach is how to select the paths to explore first. The classic solution is to use depth first exploration of the paths by backtracking [159]. However, since doing so results in an exponentially growing number of paths to be tested, the research community has proposed various heuristics to steer the execution toward unexplored regions. We discuss these techniques in Section 4.7.

4.5.2 Phase 1: learning

The aim of the learning phase is to rate the `true` and `false` directions of all conditional branches that depend on the symbolic input in the loop `L`. For each branch, we evaluate the likelihood that a particular outcome will lead to unique pointer dereferences (i.e., dereferences that we do not expect to find in the alternative outcome). Thus, we answer the question of how much we expect to gain when we follow this path, rather than the alternative. We encode this information into *weights*.

Specifically, the weights represent the likelihood of unique *access patterns*. An access pattern of the pointer `p` is the sequence of all values of `p` dereferenced during the execution of the loop. In Figure 4.1, when we denote the initial value of `u` by $\texttt{u}_0$, then the input `"//../"` triggers the following access pattern of the pointer `u`: $(\texttt{u}_0,\ \texttt{u}_0\texttt{+1},\ \texttt{u}_0 + 2,\ \texttt{u}_0\texttt{-2}, \ldots)$.

To compute the weights, we learn about the effects of individual branches. In principle, each of them may (a) directly affect the value of a pointer, (b) be a precondition for another important branch, or (c) be irrelevant from the computation's standpoint. To distinguish between these cases, *Dowser* analyzes all possible executions of a *short* symbolic input. By comparing the sets of `p`'s access patterns observed for both outcomes of a branch, it discovers which branches do not influence the diversity of pointer dereferences (i.e., are irrelevant).

Symbolic input In Section 4.4, we identified which part of the test input `I` we need to make symbolic. We denote this by $\texttt{I}_S$. In the learning phase, *Dowser* executes the loop `L` exhaustively. For performance reasons, we therefore further limit the amount of symbolic data and make only a short fragment of $\texttt{I}_S$ symbolic. For instance, for Figure 4.1, the learning phase makes only the first 4 bytes of `uri`

symbolic (not enough to trigger the bug), while scaling up to 50 symbolic bytes in the bug finding phase.

Algorithm *Dowser* exhaustively executes `L` on a short symbolic input, and records how the decisions taken at conditional branch statements influence pointer dereference instructions. For each branch `b` along the execution path, we retain the access pattern of `p` realized during this execution, `AP(p)`. We informally interpret it as "if you choose the `true` (respectively, `false`) direction of the branch `b`, expect access pattern `AP(p)` (respectively, `AP′(p)`)". This procedure results in two sets of access patterns for each branch statement, for the taken and non-taken branch, respectively. The final weight of each direction is the fraction of the access patterns that were unique for the direction in question, i.e., were not observed when the opposite one was taken.

The above description explains the intuition behind the learning mechanism, but the full algorithm is more complicated. The problem is that a conditional branch `b` might be exercised multiple times in an execution path, and it is possible that all the instances of `b` influence the access pattern observed.

Intuitively, to allow for it, we do not associate access patterns with just a single decision taken on `b` (`true` or `false`). Rather, each time `b` is exercised, we also retain which directions were previously chosen for `b`. Thus, we still collect "expected" access patterns if the `true` (respectively, `false`) direction of `b` is followed, but we augment them with a precondition. This way, when we compare the `true` and `false` sets to determine the weights for `b`, we base the scores on a deeper understanding of how an access pattern was reached.

Discussion It is important for our algorithm to avoid false negatives: we should not incorrectly flag a branch as irrelevant—it would preclude it from being explored in the bug finding phase. Say that `instr` is an instruction that dereferences the pointer `p`. To learn that a branch directly influences `instr`, it suffices to execute it. Similarly, since branches retain full access patterns of `p`, the information about `instr` being executed is also "propagated" to all its preconditions. Thus, to completely avoid false negatives, the algorithm would require full coverage of the instructions in an analysis group. We stress that we need to exercise all instructions, and not all paths in a loop. As observed by [54], exhaustive executions of even short symbolic inputs provide excellent instruction coverage in practice.

While false positives are undesirable as well, they only cause *Dowser* to execute more paths in the second phase than absolutely necessary. Due to the limited path coverage, there are corner cases, when false positives can happen. Even so, in `nginx`, only 19 out of 60 branches scored a non-zero value, which let us execute the complex loop with a 50-byte-long symbolic input.

4.5.3 Phase 2: hunting bugs

In this step, *Dowser* executes symbolically a real-world sized input in the hope of finding a value that triggers a bug. *Dowser* uses the feedback from the learning phase (Section 4.5.2) to steer its symbolic execution toward new and interesting

pointer dereferences. The goal of our heuristic is to avoid execution paths that do not bring any new pointer manipulation instructions. Thus, *Dowser* shifts the target of symbolic execution from traditional *code* coverage to *pointer value* coverage.

Dowser's strategy is explicitly dictated by the weights. As a baseline, the execution follows a depth-first exploration, and when *Dowser* is about to select the direction of a branch `b` that depends on the symbolic input, it adheres to the following rules:

- If both the `true` and `false` directions of `b` have weight 0, we do not expect `b` to influence the variety of access patterns. Thus, *Dowser* chooses the direction randomly, and does not intend to examine the other direction.
- If only one direction has a non-zero weight, we expect to observe unique access patterns only when the execution paths follows this direction, and *Dowser* favors it.
- If both of `b`'s directions have non-zero weights, both the `true` and `false` options may bring unique access patterns. *Dowser* examines both directions, and schedules them in order of their weights.

Intuitively, *Dowser*'s symbolic execution tries to select paths that are more likely to lead to overflows.

Guided fuzzing This concludes our description of *Dowser*'s architecture. To summarize, *Dowser* helps fuzzing by: (1) finding "interesting" array accesses, (2) identifying the inputs that influence the accesses, and (3) fuzzing intelligently to cover the array. Moreover, the targeted selection procedure based on pointer value coverage and the small number of symbolic input values allow *Dowser* to find bugs quickly and scale to larger applications. In addition, the ranking of array accesses permits us to zoom in on more complicated array accesses.

4.6 Evaluation

In this section, we first zoom in on the running example of `nginx` from Figure 4.1 to evaluate individual components of the system in detail (Section 4.6.1). In Section 4.6.2, we consider seven real-world applications. Based on their vulnerabilities, we evaluate our dowsing mechanism. Finally, we present an overview of the attacks detected by *Dowser*.

Since *Dowser* uses a 'spot-check' rather than 'code coverage' approach to bug detection, it must analyze each complex analysis group separately, starting with the highest ranking one, followed by the second one, and so on. Each of them runs until it finds a bug or gets terminated. The question is when we should terminate a symbolic execution run. Since symbolic execution of a single loop is highly optimized in *Dowser*, we found each bug in less than 11 minutes, so we execute each symbolic run for a maximum of 15 minutes.

Our test platform is a Linux 3.1 system with an Intel(R) Core(TM) i7 CPU clocked at 2.7GHz with 4096KB L2 cache. The system has 8GB of memory. For our experiments we used an OpenSUSE 12.1 install. We ran each test multiple times and present the median.

4.6.1 Case study: Nginx

In this section, we evaluate each of the main steps of our fuzzer by looking at our case study of `nginx` in detail.

4.6.1.1 Dowsing for candidate instructions

We measure how well *Dowser* highlights potentially faulty code and filters out the uninteresting fragments.

Our first question is whether we can filter out all the simple loops and focus on the more interesting ones. This turns out to be simple. Given the complexity scoring function from Section 4.3, we find that across all applications all analysis groups with a score less than 26 use just a single constant and at most two instructions modifying the offset of an array. Thus, in the remainder of our evaluation, we set our cut-off threshold to 26 points.

As shown in Table 4.2, `nginx` has 517 outermost loops, and only 140 analysis groups that access arrays. Thus, we throw out over 70% of the loops immediately[3]. Figure 4.5 presents the sorted weights of all the analysis groups in `nginx`. The distribution shows a quick drop after a few highly complex analysis groups. The long tail represents the numerous simple loops omnipresent in any code. 55.7% of the analysis groups score too low to be of interest. This means that *Dowser* needs to examine only the remaining 44.3%, i.e., 62 out of 140 analysis groups, or at most 12% of all loops. Out of these, the buffer overflow in Figure 4.1 ranks 4th.

4.6.1.2 Taint analysis in context of hunting for bugs

In Section 4.4 we mentioned that 'traditional' dynamic taint analysis misses implicit flows, i.e., flows that have no direct assignment of a tainted value to a variable. The problem turns out to be particularly serious for `nginx`. It receives input in text format, and transforms it to extract numerical values or various flags. As such code employs conditional statements, DTA misses the dependencies between the input and analysis groups.

Next, we evaluate the usefulness of field shifting. First, we implement the taint propagation exactly as proposed by Bao et al. [32], without any further restrictions. In that case, an index variable in the `nginx` parser becomes tainted, and we mark all HTTP fields succeeding the `uri` field as tainted as well. As a result, we introduce more symbolic data than necessary. Next, we apply field shifting (Section 4.4.2) which effectively limits taint propagation to just the `uri` field. In general, the field

[3] In principle, if a loop accesses multiple arrays, it also contains multiple access groups. Thus, these 140 analysis groups are located in fewer than 140 loops.

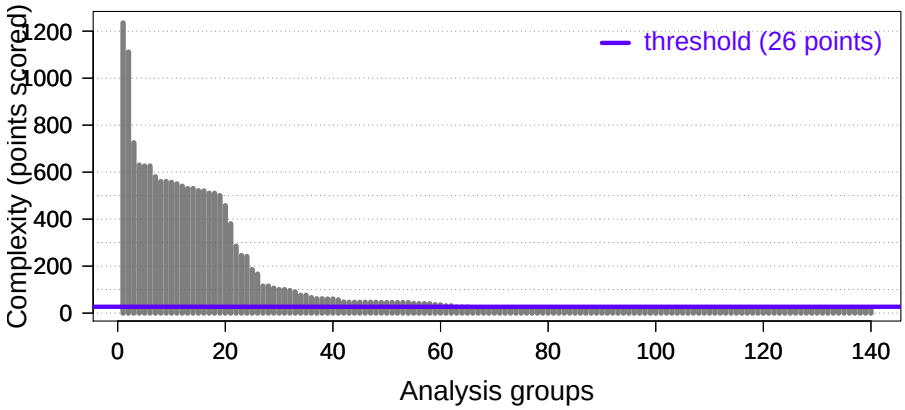


Fig. 4.5: Scores of the analysis groups in `nginx`.

shifting optimization improves the accuracy of taint propagation in all applications that take multiple input fields whose order does not matter. On the other hand, it will not help if the order is fixed.

4.6.1.3 Importance of guiding symbolic execution

We now use the `nginx` example to assess the importance of guiding symbolic execution to a vulnerability condition. For `nginx`, the input message is a generic HTTP request. Since it exercises the vulnerable loop for this analysis group, its `uri` starts with "//". Taint analysis allows us to detect that only the `uri` field is important, so we mark only this field as symbolic. As we shall see, without guidance, symbolic execution does not scale beyond very short `uri` fields (5-6 byte long). In contrast, *Dowser* successfully executes 50-byte-long symbolic `uri`s.

When S2E [71] executes a loop, it can follow one of the two search strategies: depth-first search, or maximizing code coverage (as proposed in SAGE [124]). The first one aims at complete path coverage, and the second at executing basic blocks that were not seen before. However, none can be applied in practice to examine the complex loop in `nginx`. The search is so costly that we measured the runtime for only 5-6 byte long symbolic `uri` fields. The DFS strategy handled the 5-byte-long input in 139 seconds, the 6-byte-long in 824 seconds. A 7-byte input requires more than 1 hour to finish. Likewise, the code coverage strategy required 159, and 882 seconds, respectively. The code coverage heuristic does not speed up the search for buffer overflows either, since besides executing specific instructions from the loop, memory corruptions require a very particular execution context. Even if 100% code coverage is reached, they may stay undetected.

As we explained in Section 4.5, the strategy employed by *Dowser* does not aim at full coverage. Instead, it actively searches for paths which involve new pointer dereferences. The learning phase uses a 4-byte-long symbolic input to observe ac-

cess patterns in the loop. It follows a simple depth first search strategy. As the bug clearly cannot be triggered with this input size, the search continues in the second, hunting bugs, phase. The result of the learning phase disables 66% of the conditional branches significantly reducing the exponentially of the subsequent symbolic execution. Because of this heuristic, *Dowser* easily scales up to 50 symbolic bytes and finds the bug after just a few minutes. A 5-byte-long symbolic input is handled in 20 seconds, 10 bytes in 42 seconds, 20 bytes in 63 seconds, 30 in 146 seconds, 40 in 174 seconds and 50 in 253 seconds. These numbers maintain an exponential growth of 1.1 for each added character. Even though *Dowser* still exhibits the exponential behavior, the growth rate is fairly low. Even in the presence of 50 symbolic bytes, *Dowser* quickly finds the complex bug.

In practice, symbolic execution has problems dealing with real world applications and input sizes. The number of execution paths quickly overwhelms these systems. Since triggering buffer overflows not only requires a vulnerable basic block, but also a special context, traditional symbolic execution tools are ill suited. *Dowser*, instead, requires the application to be executed symbolically for only a very short input, and then it deals with real-world input sizes instead of being limited to a few input bytes. Combined with the ability to extract the relevant parts of the original input, this enables searching for bugs in applications like web servers where input sizes were considered until now to be well beyond the scalability of symbolic execution tools.

4.6.2 Overview

In this section, we consider several applications. First, we evaluate the dowsing mechanism, and we show that it successfully highlights vulnerable code fragments. Then, we summarize the memory corruptions detected by *Dowser*. They come from six real world applications of several tens of thousands LoC, including the `ffmpeg` videoplayer of 300K LoC. The bug in `ffmpeg`, and one of the bugs in `poppler` were not documented before.

Program	Vulnerability	Dowsing		Symbolic input	Symbolic execution		
		AG score	Loops LoC		V-S2E	M-S2E	*Dowser*
nginx 0.6.32	CVE-2009-2629 heap underflow	4th out of 62/140 630 points	517 66k	URI field 50 bytes	$>$ 8 h	$>$ 8 h	253 sec
ffmpeg 0.5	UNKNOWN heap overread	3rd out of 727/1419 2186 points	1286 300k	Huffman table 224 bytes	$>$ 8 h	$>$ 8 h	48 sec
inspircd 1.1.22	CVE-2012-1836 heap overflow	1st out of 66/176 625 points	1750 45k	DNS response 301 bytes	200 sec	200 sec	32 sec
poppler 0.15.0	UNKNOWN heap overread	39th out of 388/904 1075 points	1737 120k	JPEG image 1024 bytes	$>$ 8 h	$>$ 8 h	14 sec
poppler 0.15.0	CVE-2010-3704 heap overflow	59th out of 388/904 910 points	1737 120k	Embedded font 1024 bytes	$>$ 8 h	$>$ 8 h	762 sec
libexif 0.6.20	CVE-2012-2841 heap overflow	8th out of 15/31 501 points	121 10k	EXIF tag/length 1024 + 4 bytes	$>$ 8 h	652 sec	652 sec
libexif 0.6.20	CVE-2012-2840 off-by-one error	15th out of 15/31 40 points	121 10k	EXIF tag/length 1024 + 4 bytes	$>$ 8 h	347 sec	347 sec
libexif 0.6.20	CVE-2012-2813 heap overflow	15th out of 15/31 40 points	121 10k	EXIF tag/length 1024 + 4 bytes	$>$ 8 h	277 sec	277 sec
snort 2.4.0	CVE-2005-3252 stack overflow	24th out of 60/174 246 points	616 75k	UDP packet 1100 bytes	$>$ 8 h	$>$ 8 h	617 sec

Table 4.2: Applications tested with *Dowser*. The *Dowsing* section presents the results of *Dowser*'s ranking scheme. *AG score* is the complexity of the vulnerable analysis group - its position among other analysis groups; X/Y denotes all analysis groups that are "complex enough" to be potentially analyzed/all analysis groups which access arrays; and the number of points it scores. *Loops* counts outermost loops in the whole program, and *LoC* - the lines of code according to `sloccount`. *Symbolic input* specifies how many and which parts of the input were determined to be marked as symbolic by the first two components of *Dowser*. The last section shows symbolic execution times until revealing the bug. Almost all applications proved to be too complex for the vanilla version of S2E (*V-S2E*). *Magic S2E* (*M-S2E*) is the time S2E takes to find the bug when we feed it with an input with only a minimal symbolic part (as identified in Symbolic input). Finally, the last column is the execution time of fully-fledged *Dowser*.

4.6.2.1 Dowsing for candidate instructions

We now examine several aspects of the dowsing mechanism. First, we show that there is a correlation between *Dowser*'s scoring function and the existence of memory corruption vulnerabilities. Then, we discuss how our focus on complex loops limits the search space, i.e., the amount of analysis groups to be tested. We start with a description of our data set.

Data set To evaluate the effectiveness of *Dowser*, we chose six real world programs: `nginx`, `ffmpeg`, `inspircd`, `libexif`, `poppler`, and `snort`. Additionally, we consider the vulnerabilities in `sendmail` tested by Zitser et al. [268]. For these applications, we analyzed all buffer overflows reported in CVE [189] since 2009. For `ffmpeg`, rather than include all possible codecs, we just picked the ones for which we had test cases. Out of 27 CVE reports, we took 17 for the evaluation. The remaining ten vulnerabilities are out of the scope of this paper – nine of them are related to an erroneous usage of a correct function, e.g., `strcpy`, and one was not in a loop. In this section, we consider the analysis groups from all the applications together, giving us over 3000 samples, 17 of which are known to be vulnerable[4].

When evaluating *Dowser*'s scoring mechanism, we also compare it to a straightforward scoring function that treats all instructions uniformly. For each array access, it considers exactly the same AGs as *Dowser*. However, instead of the scoring algorithm (Table 4.1), each instruction gets 10 points. We will refer to this metric as `count`.

Correlation For both *Dowser*'s and the `count` scoring functions, we computed the correlation between the number of points assigned to an analysis group and the existence of a memory corruption vulnerability. We used the Spearman rank correlation [21], since it is a reliable measure that is appropriate even when we do not know the probability distribution of the variables, or when the association between the variables is non-linear.

The positive correlation for *Dowser* is statistically significant at $p < 0.0001$, for `count` — at $p < 0.005$. The correlation for *Dowser* is stronger.

Dowsing The *Dowsing* columns of Table 4.2 shows that our focus on complex loops limits the search space from thousands of LoC to hundreds of loops, and finally to a small number of "interesting" analysis groups. Observe that `ffmpeg` has more analysis groups than loops. That is correct. If a loop accesses multiple arrays, it contains multiple analysis groups.

By limiting the analysis to complex cases, we focus on a smaller fraction of all AGs in the program, e.g., we consider 36.9% of all the analysis groups in `inspircd`, and 34.5% in `snort`. `ffmpeg`, on the other hand, contains lots of complex loops that decode videos, so we also observe many "complex" analysis groups.

[4]Since the scoring functions are application agnostic, it is sound to compare their results across applications.

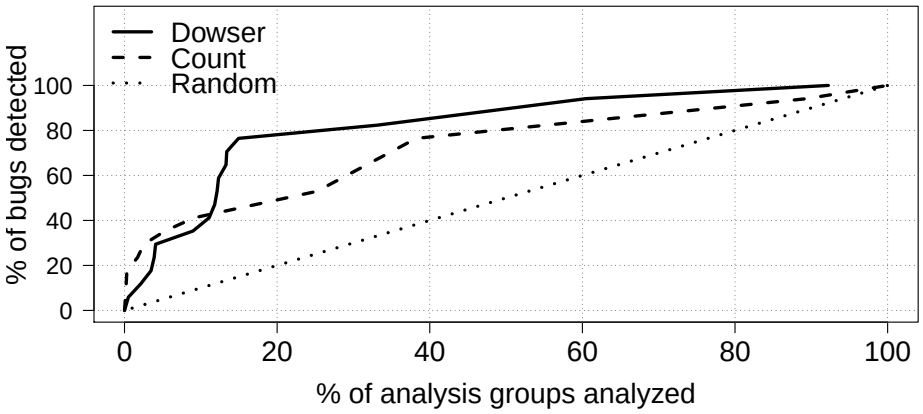


Fig. 4.6: A comparison of random testing and two scoring functions: *Dowser*'s and `count`. It illustrates how many bugs we detect if we test a particular fraction of the analysis groups.

In practice, symbolic execution, guided or not is expensive, and we can hardly afford a thorough analysis of more than just a small fraction of the target AGs of an application, say 20%-30%. For this reason, *Dowser* uses a scoring function, and tests the analysis groups in order of decreasing score. Specifically, *Dowser* looks at complexity. However, alternative heuristics are also possible. For instance, one may count the instructions that influence array accesses in an AG. To evaluate whether *Dowser*'s heuristics are useful, we compare how many bugs we discover if we examine increasing fractions of all AGs, in descending order of the score. So, we determine how many of the bugs we find if we explore the top 10% of all AGs, how many bugs we find when we explore the top 20%, and so on. In our evaluation, we are comparing the following ranking functions: (1) *Dowser*'s complexity metric, (2) counting instructions as described above, and (3) random.

Figure 4.6 illustrates the results. The random ranking serves as a baseline—clearly both `count` and *Dowser* perform better. In order to detect all 17 bugs, *Dowser* has to analyze 92.2% of all the analysis groups. However, even with just 15% of the targets, we find almost 80% (13/17) of all the bugs. At that same fraction of targets, `count` finds a little over 40% of the bugs (7/17). Overall, *Dowser* outperforms `count` beyond the 10% in the ranking. It also reaches the 100% bug score earlier than the alternatives, although the difference is minimal.

The reason why *Dowser* still requires 92% of the AGs to find *all* bugs, is that some of the bugs were very simple. The "simplest" cases include a trivial buffer overflow in `poppler` (worth 16 points), and two vulnerabilities in `sendmail` from 1999 (worth 20 points each). Since *Dowser* is designed to prioritize complex array accesses, these buffer overflows end up in the low scoring group. (The "simple"

analysis groups – with less than 26 points – start at 47.9%). Clearly, both heuristics provide much better results than random sampling. Except for the tail, they find the bugs significantly quicker, which proves their usefulness.

To summarize, we have shown that a testing strategy based on *Dowser*'s scoring function is effective. It lets us find vulnerabilities quicker than random testing or a scoring function based on the length of an analysis group.

4.6.2.2 Symbolic execution

Table 4.2 presents attacks detected by *Dowser*. The last section shows how long it takes before symbolic execution detects the bug. Since the vanilla version of S2E cannot handle these applications with the whole input marked as symbolic, we also run the experiments with minimal symbolic inputs ("*Magic* S2E"). It represents the best-case scenario when an all-knowing oracle tells the execution engine exactly which bytes it should make symbolic. Finally, we present *Dowser*'s execution times.

We run S2E for as short a time as possible, e.g., a single request/response in `nginx` and transcoding a single frame in `ffmpeg`. Still, in most applications, vanilla S2E fails to find bugs in a reasonable amount of time. `inspircd` is an exception, but in this case we explicitly tested the vulnerable DNS resolver only. In the case of `libexif`, we can see no difference between "Magic S2E" and *Dowser*, so *Dowser*'s guidance did not influence the results. The reason is that our test suite here was simple, and the execution paths reached the vulnerability condition quickly. In contrast, more complex applications process the inputs intensively, moving symbolic execution away from the code of interest. In all these cases, *Dowser* finds bugs significantly faster. Even if we take the 15 minute tests of higher-ranking analysis groups into account, *Dowser* provides a considerable improvement over existing systems.

4.7 Related work

Dowser is a 'guided' fuzzer which draws on knowledge from multiple domains. In this section, we place our system in the context of existing approaches. We start with the scoring function and selection of code fragments. Next, we discuss traditional fuzzing. We then review previous work on dynamic taint analysis in fuzzing, and finally, discuss existing work on whitebox fuzzing and symbolic execution.

Software complexity metrics Many studies have shown that software complexity metrics are positively correlated with defect density or security vulnerabilities [196, 221, 120, 266, 221, 200]. However, Nagappan et al. [196] argued that no single set of metrics fits all projects, while Zimmermann et al. [266] emphasize a need for metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns. All these approaches consider the broad class of post-release defects or security vulnerabilities, and consider a very generic set of

measurements, e.g., the number of basic blocks in a function's control flow graph, the number of global or local variables read or written, the maximum nesting level of `if` or `while` statements and so on. *Dowser* is very different in this respect, and to the best of our knowledge, the first of its kind. We focus on a narrow group of security vulnerabilities, i.e., buffer overflows, so our scoring function is tailored to reflect the complexity of pointer manipulation instructions.

Traditional fuzzing Software fuzzing started in earnest in the 90s when Miller et al. [186] described how they fed random inputs to (UNIX) utilities, and managed to crash 25-33% of the target programs. More advanced fuzzers along the same lines, like Spike [231], and SNOOZE [31], deliberately generate malformed inputs, while later fuzzers that aim for deeper bugs are often based on the input grammar (e.g., Kaksonen [151] and [239]). DeMott [97] offers a survey of fuzz testing tools. As observed by Godefroid et al. [124], traditional fuzzers are useful, but typically find only shallow bugs.

Application of DTA to fuzzing BuzzFuzz [119] uses DTA to locate regions of seed input files that influence values used at library calls. They specifically select library calls, as they are often developed by different people than the author of the calling program and often lack a perfect description of the API. Buzzfuzz does not use symbolic execution at all, but uses DTA only to ensure that they preserve the right input format. Unlike *Dowser*, it ignores implicit flows completely, so it could never find bugs such as the one in nginx (Figure 4.1). In addition, *Dowser* is more selective in the application of DTA. It's difficult to assess which library calls are important and require a closer inspection, while *Dowser* explicitly selects complex code fragments.

TaintScope [250] is similar in that it also uses DTA to select fields of the input seed which influence security-sensitive points (e.g., system/library calls). In addition, TaintScope is capable of identifying and bypassing checksum checks. Like Buzzfuzz, it differs from *Dowser* in that it ignores implicit flows and assumes only that library calls are the interesting points. Unlike BuzzFuzz, TaintScope operates at the binary level, rather than the source.

Symbolic-execution-based fuzzing Recently, there has been much interest in white-box fuzzing, symbolic execution, concolic execution, and constraint solving. Examples include EXE [55], KLEE [54], CUTE [219], DART [121], SAGE [124], and the work by Moser et al. [194]. Microsoft's SAGE, for instance, starts with a well-formed input and symbolically executes the program under test in attempt to sweep through all feasible execution paths of the program. While doing so, it checks security properties using AppVerifier. All of these systems substitute (some of the) program inputs with symbolic values, gather input constraints on a program trace, and generate new input that exercises different paths in the program. They are very powerful, and can analyze programs in detail, but it is difficult to make them scale (especially if you want to explore many loop-based array accesses). The problem is that the number of paths grows very quickly.

Zesti [177] takes a different approach and executes existing regression tests symbolically. Intuitively, it checks whether they can trigger a vulnerable condition by slightly modifying the test input. This technique scales better and is useful for finding bugs in paths in the neighborhood of existing test suites. It is not suitable for bugs that are far from these paths. As an example, a generic input which exercises the vulnerable loop in Figure 4.1 has the `uri` of the form "//{*arbitrary characters*}", and the shortest input triggering the bug is "//../". When fed with "`//abc`", [177] does not find the bug—because it was not designed for this scenario. Instead, it requires an input which is much closer to the vulnerability condition, e.g., "//..{*an arbitrary character*}". For *Dowser*, the generic input is sufficient.

SmartFuzz [192] focuses on integer bugs. It uses symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. In contrast, *Dowser* targets the more common (and harder to find) case of buffer overflows. Finally, Babić et al. [28] guide symbolic execution to potentially vulnerable program points detected with static analysis. However, the interprocedural context- and flow-sensitive static analysis proposed does not scale well to real world programs and the experimental results contain only short traces.

4.8 Conclusion

Dowser is a guided fuzzer that combines static analysis, dynamic taint analysis, and symbolic execution to find buffer overflow vulnerabilities deep in a program's logic. It starts by determining 'interesting' array accesses, i.e., accesses that are most likely to harbor buffer overflows. It ranks these accesses in order of complexity—allowing security experts to focus on complex bugs, if so desired. Next, it uses taint analysis to determine which inputs influence these array accesses and fuzzes only these bytes. Specifically, it makes (only) these bytes symbolic in the subsequent symbolic execution. Where possible *Dowser*'s symbolic execution engine selects paths that are most likely to lead to overflows. Each three of the steps contain novel contributions in and of themselves (e.g., the ranking of array accesses, the implicit flow handling in taint analysis, and the symbolic execution based on pointer value coverage), but the overall contribution is a new, practical and complete fuzzing approach that scales to real applications and complex bugs that would be hard or impossible to find with existing techniques. Moreover, *Dowser* proposes a novel 'spot-check' approach to finding buffer overflows in real software.

5

Body armor for binaries: preventing buffer overflows without recompilation

Preamble: Relation to the Research Roadmap

While the previous chapter discussed buffer overflow vulnerabilities and how to find them, this chapter is concerned with detecting such attacks as they occur. As a result, it is related to the same themes in the Red Book. Specifically, it has a link to Chapters 4 and 9 ("Software Vulnerabilities" and "Legacy Systems") as BinArmor tries to detect buffer overflows attacks immediately (as soon as they occur) and operates on the binary. Moreover, it is, to our knowledge, unique in that it can stop extremely advanced buffer overflow attacks—namely those that do not divert the program's control flow, but rather change its data.

Like the previous chapter, the current chapter also relates to Chapter 6 of the Red Book ("Critical Infrastructures"), as many of these systems contain legacy code. The source code for such systems may not be available, so that hardening such legacy code at the binary level is the only option. Again, the notorious Stuxnet attack on the uranium enrichment facility in Iran in 2010, made use of a fairly classic memory corruption attack (among several others).

Asia Slowinska, Traian Stancescu, Herbert Bos, "Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation" 2012 USENIX Annual Technical Conference (USENIX ATC 12), Boston, USA – August 2012.

Abstract

BinArmor is a novel technique to protect existing C binaries from memory corruption attacks on both control data and non-control data. Without access to source code, non-control data attacks cannot be detected with current techniques. Our approach hardens binaries against both kinds of overflow, without requiring the programs' source or symbol tables. We show that *BinArmor* is able to stop real attacks—including the recent non-control data attack on Exim. Moreover, we did not incur a single false positive in practice. On the downside, the current overhead of *BinArmor* is high—although no worse than competing technologies like taint analysis that do not catch attacks on non-control data. Specifically, we measured an overhead of 70% for `gzip`, 16%-180% for `lighttpd`, and 190% for the `nbench` suite.

5.1 Introduction

Despite modern security mechanisms like stack protection [80], ASLR [41], and PaX/DEP/W⊕X [240], buffer overflows rank third in the CWE SANS top 25 most dangerous software errors [90]. The reason is that attackers adapt their techniques to circumvent our defenses.

Non-control data attacks, such as the well-known attacks on *exim* mail servers (Section 5.2), are perhaps most worrying [67, 230]. Attacks on non-control data are hard to stop, because they do not divert the control flow, do not execute code injected by the attacker, and often exhibit program behaviors (e.g., in terms of system call patterns) that may well be legitimate. Worse, for binaries, we do not have the means to detect them *at all*.

Current defenses against non-control data attacks all require access to the source code [149, 22, 23]. In contrast, security measures at the binary level can stop various control-flow diversions [75, 20, 108], but offer no protection against corruption of non-control data.

Even for more traditional control-flow diverting attacks, current binary instrumentation systems detect only the *manifestations* of attacks, rather than the attacks themselves. For instance, they detect a control flow diversion that *eventually* results from the buffer overflow, but not the actual overflow itself, which may have occurred thousands of cycles before. The lag between time-of-attack and time-of-manifestation makes it harder to analyze the attack and find the root cause [225].

In this paper, we describe *BinArmor*, a tool to bolt a layer of protection on C binaries that stops state-of-the-art buffer overflows immediately (as soon as they occur).

High level overview Rather than patching systems after a vulnerability is found, *BinArmor* is proactive and stops buffer (array) overflows in binary software, before we even know it is vulnerable. Whenever it detects an attack, it will raise an alarm and abort the execution. Thus, like most protection schemes, we assume

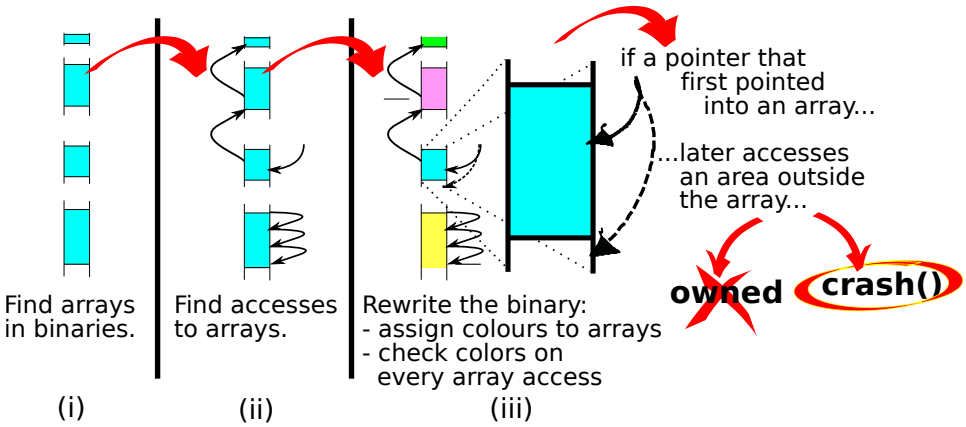


Fig. 5.1: *BinArmor* overview.

that the system can tolerate rare crashes. Finally, *BinArmor* operates in one of two modes. In *BA-fields mode*, we protect individual fields inside structures. In *BA-objects mode*, we protect at the coarser granularity of full objects.

BinArmor relies on limited information about the program's data structures—specifically the buffers that it should protect from overflowing. If the program's symbol tables are available, *BinArmor* is able to protect the binary against buffer overflows with great precision. Moreover, in BA-objects mode no false positives are possible in this case. While we cannot guarantee this in BA-fields mode, we did not encounter any false positives in practice, and as we will discuss later, they are unlikely.

However, while researchers in security projects frequently assume the availability of symbol tables [108], in practice, software vendors often strip their code of all debug symbols. In that case, we show that we can use automated reverse engineering techniques to extract symbols from stripped binaries, and that this is enough to protect real-world applications against real world-attacks. To our knowledge, we are the first to use data structure recovery to prevent memory corruption. We believe the approach is promising and may also benefit other systems, like XFI [108] and memory debuggers [198].

BinArmor hardens C binaries in three steps (Fig. 5.1):

(i) *Data structure discovery:* dynamically extract the data structures (buffers) that need protection.

(ii) *Array access discovery:* dynamically find potentially unsafe pointer accesses to these buffers.

(iii) *Rewrite:* statically rewrite the binary to ensure that a pointer accessing a buffer stays within its bounds.

Data structure discovery is easy when symbol tables are available, but very hard when they are not. In the absence of symbol tables, *BinArmor* uses recent research

results [227] to reverse engineer the data structures (and especially the buffers) from the binary itself by analyzing memory access patterns (Fig. 5.1, step i). Something is a struct, if it is accessed like a struct, and an array, if it is accessed like an array. And so on. Next, given the symbols, *BinArmor* dynamically detects buffer accesses (step ii). Finally, in the rewrite stage (step iii), it takes the data structures and the accesses to the buffers, and assigns to each buffer a unique color. Every pointer used to access the buffer for the first time obtains the color of this buffer. *BinArmor* raises an alert whenever, say, a blue pointer accesses a red byte.

Contributions *BinArmor* proactively protects existing C binaries, before we even know whether the code is vulnerable, against attacks on control data *and* non-control data, and it can do so either at object or sub-field granularity. Compared to source-level protection like WIT, *BinArmor* has the advantage that it requires *no access to source code or the original symbol tables*. In addition, in BA-fields mode, by *protecting individual fields* inside a structure rather than aggregates, *BinArmor* is finer-grained than WIT and similar solutions. Also, it prevents overflows on both writes *and* reads, while WIT protects only writes and permits information leakage. Further, we show in Section 5.9 that points-to analysis (a technique relied on by WIT), is frequently imprecise.

Compared to techniques like taint analysis that also target *binaries*, *BinArmor* detects both control flow and *non-control flow attacks*, whereas taint analysis detects only the former. Also, it detects attacks *immediately* when they occur, rather than sometime later, when a function pointer is used.

The main drawback of *BinArmor* is the very significant slowdown (up to 2.8x for the lighttpd webserver and 1.7x for gzip). While better than most tainting systems (which typically incur 3x-20x), it is much slower than WIT (1.04x for gzip). Realistically, such slowdowns make *BinArmor* in its current form unsuitable for any system that requires high performance. On the other hand, it may be used in application domains where security rather than performance is of prime importance. In addition, because *BinArmor* detects buffer overflows themselves rather than their manifestations, we expect it to be immediately useful for security experts analyzing attacks. Finally, we will show later that we have not explored all opportunities for performance optimization.

Our work builds on dynamic analysis, and thus suffers from the limitations of all dynamic approaches: we can only protect what we execute during the analysis. This work is *not* about code coverage. We rely on existing tools and test suites to cover as much of the binary as possible. Since coverage is never perfect, we may miss buffer accesses and thus incur false negatives. Despite this, *BinArmor* detected all 12 real-world buffer overflow attacks in real-world applications we study (Section 5.8).

BinArmor takes a conservative approach to prevent false positives (unnecessary program crashes). For instance, no false positives are possible when the protection is limited to structures (BA-objects mode). In BA-fields mode, we can devise sce-

narios that lead to false positives due to the limited code coverage. However, we did not encounter any in practice, and we will show that they are very unlikely.

Since our dynamic analysis builds on Qemu [37] process emulation which is only available for Linux, we target x86 Linux binaries, generated by `gcc` (albeit of various versions and with different levels of optimization). However, there is nothing fundamental about this and the techniques should apply to other systems also.

5.2 Some buffer overflows are hard to stop: the Exim attack on non-control data

In December 2010, Sergey Kononenko posted a message on the *exim* developers mailing list about an attack on the *exim* mail server. The news was slashdotted shortly after. The remote root vulnerability in question concerns a heap overflow that causes adjacent heap variables to be overwritten, for instance an access control list (ACL) for the sender of an e-mail message. A compromised ACL is bad enough, but in *exim* the situation is even worse. Its powerful ACL language can invoke arbitrary Unix processes, giving attackers full control over the machine.

The attack is a typical heap overflow, but what makes it hard to detect is that it does not divert the program's control flow at all. It only overwrites non-control data. ASLR, W⊕X, canaries, system call analysis—all fail to stop or even detect the attack.

Both 'classic' buffer overflows [106], and attacks on non-control data [67] are now mainstream. While attackers still actively use the former (circumventing existing measures), there is simply *no* practical defense against the latter in binaries. Thus, researchers single out non-control data attacks as a serious future threat [230]. *BinArmor* protects against both types of overflows.

5.3 What to Protect: Buffer Accesses

BinArmor protects binaries by instrumenting buffer accesses to make sure they are safe from overflows. Throughout the paper, *a buffer* is an array that can potentially overflow. Fig. 5.1 illustrates the general idea, which is intuitively simple: once the program has assigned an array to a pointer, it should not use the same pointer to access elements beyond the array bounds. For this purpose, *BinArmor* assigns colors to arrays and pointers and verifies that the colors of memory and pointer match on each access. After statically rewriting the binary, the resulting code runs natively and incurs overhead only for the instructions that access arrays. In this section, we explain how we obtain buffers and accesses to them when symbols are not available, while Sections 5.5–5.7 discuss how we use this information to implement fine-grained protection against buffer overflows.

5.3.1 Extracting Buffers and Data Structures

Ideally, *BinArmor* obtains information about buffers from the symbol tables. Many projects assume the availability of symbol tables [108, 198]. Indeed, if the binary does come with symbols, *BinArmor* offers very accurate protection. However, as symbols are frequently stripped off in real software, it uses automated reverse engineering techniques to extract them from the binary. *BinArmor* uses a dynamic approach, as static approaches are weak at recovering arrays, but, in principle, they work also [213].

Specifically, we recover arrays using Howard [227], which follows the simple intuition that memory access patterns reveal much about the layout of data structures. In this paper, we sketch only the general idea and refer to the original Howard paper for details [227]. Using binary code coverage techniques [71, 54], Howard executes as many of the execution paths through the binary as possible and observes the memory accesses. To detect arrays, it first detects loops and then treats a memory area as an array if (1) the program accesses the area in a loop (either consecutively, or via arbitrary offsets from the array base), and (2) all accesses 'look like' array accesses (e.g., fixed-size elements). Moreover, it takes into account array accesses outside the loop (including 'first' and 'last' elements), and handles a variety of complications and optimizations (like loop unrolling).

Since arrays are detected dynamically, we should not underestimate the size of arrays, lest we incur false positives. If the array is classified as too small, we might detect an overflow when there is none. In Howard, the data structure extraction is deliberately conservative, so that in practice the size of arrays is either classified exactly right, or overestimated (which never leads to false positives). The reason is that it conservatively extends arrays towards the next variable below or above. Howard is very unlikely to underestimate the array size for compiler-generated code and we never encountered it in any of our tests, although there is no hard guarantee that we never will. Size underestimation is possible, but can happen only if the program accesses the array with multiple base pointers, and behaves consistently and radically different in all analysis runs from the production run.

Over a range of applications, Howard never underestimated an array's size and classified well over 80% of all arrays on the executed paths 'exactly right'—down to the last byte. These arrays represent over 90% of all array bytes. All remaining arrays are either not classified at all or overestimated and thus safe with respect to false positives.

We stressed earlier that Howard aims to err on the safe side, by overestimating the size of arrays to prevent false positives. The question is what the costs are of doing so. Specifically, one may expect an increase in false negatives. While true in theory, this is hardly an issue in practice. The reason is that *BinArmor* only misses buffer overflows that (1) overwrite values immediately following the real array (no byte beyond the (over-)estimation of the array is vulnerable), *and* (2) that overwrite a value that the program did not use separately during the dynamic analysis of the program (otherwise, we would not have classified it as part of the array).

Exploitable overflows that satisfy both conditions are rare. For instance, an overflow of a return value would never qualify, as the program always uses the return address separately. Overall, not a single vulnerability in Linux programs for which we could find an exploit qualified.

One final remark about array extraction and false positives; as mentioned earlier, *BinArmor* does not care which method is used to extract arrays and static extractors may be used just as well. However, this is not entirely true. Not underestimating array sizes is crucial. We consider the problem of finding correct buffer sizes orthogonal to the binary protection mechanism offered by *BinArmor*. Whenever we discuss false positives in *BinArmor*, *we always assume that the sizes of buffers are not underestimated*.

5.3.2 Instructions to be Instrumented

When *BinArmor* detects buffers to be protected, it dynamically determines the instructions (array accesses), that need instrumenting. The process is straightforward: for each buffer, it dumps all instructions that access it.

Besides accesses, *BinArmor* also dumps all instructions that initialize or manipulate pointers that access arrays.

5.4 Code Coverage and Modes of Operation

Since *BinArmor* is based on dynamic analysis, it suffers from coverage issues—we can only analyze what we execute. Even the most advanced code coverage tools [54, 71] cover just a limited part of real programs. Lack of coverage causes *BinArmor* to miss arrays and array accesses and thus incur false negatives. Even so, *BinArmor* proved powerful enough to detect *all* attacks we tried (Section 5.8). What we really want to avoid are false positives: crashes on benign input.

In *BinArmor*, we instrument only those instructions that we encountered during the analysis phase. However, a program path executed at runtime, p_R, may differ from all paths we have seen during analysis A, $\{p_a\}_{a \in A}$, and yet p_R might share parts with (some of) them. Thus, an arbitrary subset of array accesses and pointer manipulations on p_R is instrumented, and as we instrument exactly those instructions that belong to paths in $\{p_a\}_{a \in A}$, it may well happen that we miss a pointer copy, a pointer initialization, or a pointer dereference instruction.

With that in mind, we should limit the color checks performed by *BinArmor* to program paths which use array pointers in ways also seen during analysis. Intuitively, the more scrupulous and fine-grained the color checking policy, the more tightly we need to constrain protected program paths to the ones *seen before*. To address this tradeoff, we offer two modes of *BinArmor* which impose different requirements for the selection of program paths to be instrumented, and offer protection at different granularities: coarse-grained *BA-objects* mode (Section 5.5), and fine-grained *BA-fields* mode (Section 5.6).

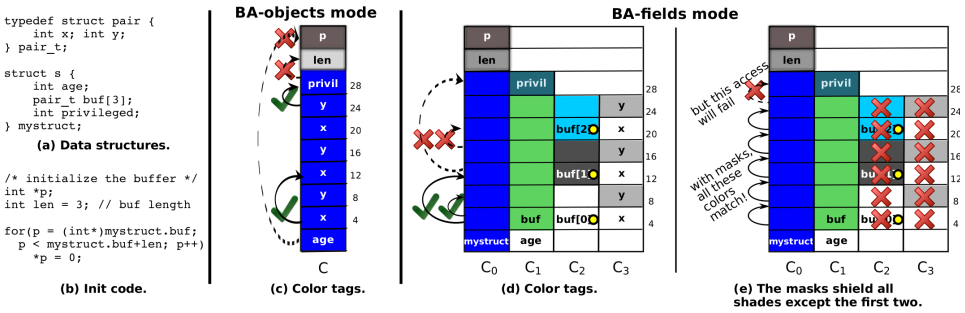


Fig. 5.2: *BinArmor* colors in BA-objects mode (c) and BA-fields modes (d,e) for sample data structures (a) and code (b).

5.5 *BA-objects mode*: Object-level Protection

Just like other popular approaches, e.g., WIT [22] and BBC [23], BA-objects mode provides protection at the level of objects used by a program. To do so, *BinArmor* assigns a color to each buffer[1] on stack, heap, or in global memory. Then it makes sure that a pointer to an object of color X never accesses memory of color Y. This way we detect all buffer overflows that aim to overwrite another object in memory.

5.5.1 What is Permissible? What is not?

Figs. (5.2.a-5.2.b) show a function with some local variables, and Fig. (5.2.c) shows their memory layout and colors. In BA-objects mode, we permit memory accesses within objects, such as the two tick-marked accesses in Fig. (5.2.c). In the first case, the program perhaps iterates over the elements in the array (at offsets 4, 12, and 20 in the object), and dereferences a pointer to the second element (offset 12) by adding `sizeof(pair_t)` to the array's base pointer at offset 4. In the second case, it accesses the `privileged` field of `mystruct` via a pointer to the last element of the array (offset 24). Although the program accesses a field beyond the array, it remains within the local variable `mystruct`, and (like WIT and other projects), we allow such operations in this mode. Such access patterns commonly occur, e.g., when a `memset()`-like function initializes the entire object.

However, *BinArmor* stops the program from accessing the `len` and `p` fields through a pointer into the structure. `len`, `p` and `mystruct` are separate variables on the stack, and one cannot be accessed through a pointer to the other. Thus, *BinArmor* in BA-objects mode stops inter-object buffer overflow attacks, but not intra-object ones.

5.5.2 Protection by Color Matching

BinArmor uses colors to enforce protection. It assigns colors to each word of a buffer[1], when the program allocates memory for it in global, heap, or stack memory. Each complete object gets one unique color. All memory which we do not protect gets a unique background color.

When the program assigns a buffer of color X to a pointer, *BinArmor* associates the same color with the register containing the pointer. The color does not change when the pointer value is manipulated (e.g., when the program adds an offset to the pointer), but it is copied when the pointer is copied to a new register. When the pointer is stored to memory, we also store its color to a memory map, to load it later when the pointer is restored.

From now on, *BinArmor* vets each dereference of the pointer to see if it is still in bounds. Vetting pointer dereferences is a matter of checking whether the color of the pointer matches that of the memory to which it points.

Stale Colors and Measures to Rule out False Positives Due to lack of coverage, a program path at runtime may lack instrumentation on some pointer manipulation instructions. This may lead to the use of a *stale* color.

Consider a function like `memcpy(src,dst)`. Suppose that *BinArmor* misses the `dst` buffer during analysis (it was never used), so that it (erroneously) does not instrument the instructions manipulating the `dst` pointer prior to calling `memcpy()`—say, the instruction that pushes `dst` on the stack. Also suppose that `memcpy()` itself *is* instrumented, so the load of the `dst` pointer into a register obtains the color of that pointer. However, since the original push was not instrumented, *BinArmor* never set that color! If we are lucky, we simply find no color, and everything works fine. If we are unlucky, we pick up a stale color of whatever was previously on the stack at that position[2]. As soon as `memcpy()` dereferences the pointer, the color check fails and the program crashes.

BinArmor removes all false positives of this nature by adding an additional tag to the colors to indicate to which memory address the color corresponds. The tag functions not unlike a tag in a hardware cache entry: to check whether the value we find really corresponds to the address we look for. For instance, if `eax` points to `dst`, the tag contains the address `dst`. If the program copies `eax` to `ebx`, it also copies the color and the tag. When the program manipulates the register (e.g., `eax++`), the tag incurs the same manipulation (e.g., tag_{eax}++). Finally, when the program dereferences the pointer, we check whether the color corresponds to the memory to which the pointer refers. Specifically, *BinArmor* checks the colors on a dereference of `eax`, **iff** *(tag_{eax}==eax)*. Thus, it ignores stale colors and prevents the false positives.

[1] Or a struct containing the array as this mode operates on objects

[2] There may be stale colors for the stack value, because it is not practical to clean up all colors whenever memory is no longer in use.

Pointer Subtraction: What if Code is Color Blind? The colors assigned by *BinArmor* prevent a binary from accessing object X though a pointer to object Y. Even though programs in C are not expected to do so, some functions exhibit "color blindness", and directly use a pointer to one object to access another object. The `strcat()` and `__strcpy_chk()` functions in current libc implementations on Linux are the best known examples: to copy a source to a destination string, they access both by the same pointer—adding the *distance* between them to access the remote string.

Our current solution is straightforward. When *BinArmor* detects a pointer subtraction, and later spots when the resultant distance is added to the subtrahend to access the buffer associated with the minuend pointer, it resets the color to reflect the remote buffer, and we protect dereferences in the usual way.

If more complex implementations of this phenomenon appear, we can prevent the associated dereferences from being checked at all. To reach the remote buffer, such scenarios have an operation which involves adding a value derived from the distance between two pointers. *BinArmor* would not include it in the set of instructions to be instrumented, so that the tag of the resultant pointer will not match its value, and the color check will not be performed. False positives are ruled out.

Other projects, like WIT [22] and the pointer analysis-based protection in [27], explicitly assume that a pointer to an object can only be derived from a pointer to the same object. In this sense, our approach is more generic.

5.5.3 Expect the Unexpected Paths

To justify that *BinArmor* effectively rules out false positives, we have to show that all program paths executed at runtime do not exhibit any false alerts. As we discussed in Section 5.4, a program path at runtime, p_R, may differ from all paths seen during analysis, while sharing parts with (some of) them. Thus, p_R may access an array, while some of the instructions associated with these accesses are not instrumented. The question is whether p_R may cause false positives.

Suppose p_R accesses an array. If `arr` is a pointer to this array, 3 generic types of instruction might be missed, and thus not instrumented by *BinArmor*: (1) an `arr` initialization instruction, (2) an `arr` update/manipulation instruction, and (3) an `arr` dereference instruction.

The crucial feature of *BinArmor* which prevents false positives in cases (1) and (2) are the tags introduced in Section 5.5.2. They check whether the color associated with a pointer corresponds to the right value. In the case of a pointer initialization or a pointer update instruction missing, the pointer tag does not match its value anymore, its color is considered invalid, and it is not checked on dereferences. Finally, if an `arr` dereference instruction is not instrumented, it only means that the color check is not performed. Again, it can only result in false negatives, but never false positives.

5.6 *BA-fields mode*: a Colorful Armor

BA-objects mode and BA-fields mode differ significantly in the granularity of protection. Where BA-objects mode protects memory at the level of objects, BA-fields mode offers finer-grained protection—at the level of fields in structures. Thus, BinArmor in BA-fields mode stops not only inter-object buffer overflow attacks, but also intra-object ones. We shall see, the extra protection increases the chances of false positives which should be curbed.

5.6.1 What is Permissible? What is not?

Consider the structure in Fig. (5.2.a) with a memory layout as shown in Fig. (5.2.d). Just like in BA-objects mode, *BinArmor* now also permits legitimate memory accesses such as the two tick-marked accesses in Fig. (5.2.d).

But unlike in BA-objects mode, *BinArmor* in BA-fields mode stops the program from accessing the `privileged` field *via* a pointer into the array. Similarly, it prevents accessing the `x` field in one array element from the `y` field in another. Such accesses that do not normally occur in programs are often symptomatic of attacks[3].

5.6.2 Shaded Colors

BinArmor uses a *shaded* color scheme to enforce fine-grained protection. Compared to BA-objects mode, the color scheme used here is much richer. In Section 5.5, the whole object was given a single color, but in BA-fields mode, we add shades of colors to distinguish between individual fields in a structure. First, we sketch how we assign the colors. Next, we explain how they are used.

Since *BinArmor* knows the structure of an object to be protected, it can assign separate colors to each variable and to each field. The colors are hierarchical, much like real colors: lime green is a shade of green, and dark lime green and light lime green, are gradations of lime green, etc. Thus, we identify a byte's color as a sequence of shades: $C_0 : C_1 : .. : C_N$, where we interpret C_{i+1} as a shade of color C_i. Each shade corresponds to a nesting level in the data structure. This is illustrated in Fig. (5.2.d).

The base color, C_0, corresponds to the complete object, and is just like the color used by *BinArmor* in BA-objects mode. It distinguishes between individually allocated objects. At level 1, the object in Fig. (5.2.d) has three fields, each of which gets a unique shade C_1. The two integer fields do not have any further nesting, but the array field has two more levels: array elements and fields within the array elements. Again, we assign a unique shade to each array element and, within each array element, to each field. The only exceptions are the base of the array and the base of the structs—they remain blank for reasons we explain shortly. Finally, each

[3]Note: if they *do* occur, either `Howard` classifies the data structures differently, or *BinArmor* detects these accesses in the analysis phase, and applies *masks* (Section 5.6.2), so they do not cause problems.

color C_i has a type flag indicating whether it is an array element shown in the figure as a dot (a small circle on the right).

We continue the coloring process, until we reach the maximum nesting level (in the figure, this happens at C_3), or exhaust the maximum color depth N. In the latter case, the object has more levels of nesting than *BinArmor* can accommodate in shades, so that some of the levels will collapse into one, 'flattening' the substructure. Collapsed structures reduce *BinArmor*'s granularity, but do not cause problems otherwise. In fact, most existing solutions (like WIT [22] and BBC [23]) operate only at the granularity of the full object.

Protection by Color Matching The main difference between the color schemes implemented in BA-objects mode and BA-fields mode is that colors are more complex now and include multiple shades. We need a new procedure to compare them, and decide what is legal.

The description of the procedure starts in exactly the same way as in BA-objects mode. When a buffer of color X is assigned to a pointer, *BinArmor* associates the same color with the register containing the pointer. The color does not change when the pointer value is manipulated (e.g., when the program adds an offset to the pointer), but it is copied when the pointer is copied to a new register. When the program stores a pointer to memory, we also store its color to a memory map, to load it later when the pointer is restored to a register.

The difference from the BA-objects mode is in the color update rule: when the program dereferences a register, we update its color so that it now corresponds to the memory location associated with the register. The intuition is that we do not update colors on intermediate pointer arithmetic operations, but that the colors represent pointers used by the program to access memory.

From now on, *BinArmor* vets each dereference of the pointer to see if it is still in bounds. Vetting pointer dereferences is a matter of checking whether the color of the pointer matches that of the memory it points to—in all the shades, from left to right. Blank shades serve as wild cards and match any color. Thus, leaving bases of structures and arrays blank guarantees that a pointer to them can access all internal fields of the object.

Finally, we handle the common case where a pointer to an array element derives from a pointer to another element of the array. Since array elements in Fig. (5.2c) differ in C_2, such accesses would normally not be allowed, but the dots distinguish array elements from structure fields. Thus we are able to grant these accesses. We now illustrate these mechanisms for our earlier examples.

Suppose the program has already accessed the first array element by means of a pointer to the base of the array at offset 4 in the object. In that case, the pointer's initial color is set to C_1 of the array's base. Next, the program adds `sizeof(pair_t)` to the array's base pointer and dereferences the result to access the second array element. At that point, *BinArmor* checks whether the colors match. C_0 clearly matches, and since the pointer has only the C_1 color of the first array element, its

color and that of the second array element match. Our second example, accessing the `y` field from the base of the array, matches for the same reason.

However, an attacker cannot use this base pointer to access the `privileged` field, because the C_1 colors do not match. Similarly, going from the `y` field in the second array element to the `x` field in the third element will fail, because the C_2 shades differ.

The Use of Masks: What if Code is Color Blind? Programs do not always access data structures in a way that reflects the structure. They frequently use functions similar to `memset` to initialize (or copy) an entire object, with all subfields and arrays in it. Unfortunately, these functions do not heed the structure at all. Rather, they trample over the entire data structure in, say, word-size strides. Here is an example. Suppose `p` is a pointer to an integer and we have a custom `memset`-like function:

```
for (p=objptr, p<sizeof(*objptr); p++) *p = 0;
```

The code is clearly 'color blind', but while it violates the color protection, *BinArmor* should not raise an alert as the accesses are all legitimate. But it should not ignore color blindness either. For instance, the initialization of one object should not trample over *other* objects. Or inside the structure of Fig. (5.2.b): an initialization of the array should not overwrite the `privileged` field.

One (bad) way to handle such color blindness is to white-list the code. For instance, we could ignore all accesses from white-listed functions. While this helps against some false alerts, it is not a good solution for two reasons. First, it does not scale; it helps only against a few well-known functions (e.g., libc functions), but not against applications that use custom functions to achieve the same. Second, as it ignores these functions altogether, it would miss attacks that use this code. For instance, the initialization of (just) the buffer could overflow into the privilege field.

Instead, *BinArmor* exploits the shaded colors of Section 5.6.2 to implement *masks*. Masks shield code that is color blind from some of the structure's subtler shades. For instance, when the initialization code in Fig. (5.2.b) is applied to the array, we filter out all shades beyond C_1: the code is then free to write over all the records in the array, but cannot write beyond the array. Similarly, if an initialization routine writes over the entire object, we filter all shades except C_0, limiting all writes to this object.

Fig. (5.2.e) illustrates the usage of masks. The code on the left initializes the array in the structure of Fig. 5.2. By masking all colors beyond C_0 and C_1, all normal initialization code is permitted. If attackers can somehow manipulate the `len` variable, they could try to overflow the buffer and change the `privileged` value. However, in that case the C_1 colors do not match, and *BinArmor* will abort the program.

To determine whether a memory access needs masks (and if so, what sort), *BinArmor*'s dynamic analysis first marks all instructions that trample over multiple

data structures as 'color blind' and determines the appropriate mask. For instance, if an instruction accesses the base of the object, *BinArmor* sets the masks to block out all colors except C_0. If an instruction accesses a field at the kth level in the structure, *BinArmor* sets the masks to block out all colors except $C_0 ... C_k$. And so on.

Finding the right masks to apply and the right places to do so, requires fairly subtle analysis. *BinArmor* needs to decide *at runtime* which part of the shaded color to mask. In the above example, if the program initializes the whole structure, *BinArmor* sets the masks to block out all colors except C_0. If the same function is called to initialize the array, however, only C_2 and C_3 are shielded. To do so, *BinArmor*'s dynamic analysis tracks the *source* of the pointer used in the 'color blind' instruction, i.e., the base of the structure or array. The instrumentation then allows for accesses to all fields included in the structure (or substructure) rooted at this source. Observe that not all such instructions need masks. For instance, code that zeros all words in the object by adding increasing offsets to the *base* of the object, has no need for masks. After all, because of the blank shades the base of the object permits access to the entire object even without masks.

BinArmor enforces the masks when rewriting the binary. Rather than checking all shades, it checks only the instructions' *visible* colors for these instructions.

Pointer Subtraction As discussed in Section 5.5.2, some functions exhibit color blindness, and use a pointer to one object to access another. Both the problem and its solution are exactly the same as for BA-fields mode.

5.6.3 Why We do Not See False Positives

Given an accurate or conservative estimate of array sizes, the only potential cause of false positives is lack of coverage. As explained in Section 5.5, we do not address the array size underestimation here–we simply require either symbol tables or a conservative data structure extractor (Section 5.3). But other coverage issues occur regardless of symbol table availability and must be curbed.

Stale Colors and Tags In Section 5.5.2, we showed that lack of coverage could lead to the use of stale colors in BA-objects mode. Again, the problem and its solution are the same as for BA-fields mode.

Missed Masks and Context Checks Limited code coverage may also cause *BinArmor* to miss the *need* for masks and, unless prevented, lead to false positives. Consider again the example custom `memset` function of Section 5.6.2. The code is color blind, unaware of the underlying data structure, and accesses the memory according to its own pattern. To prevent false positives, we introduced *masks* that filter out some shades to allow for benign memory accesses.

Suppose that during analysis the custom `memset` function is invoked only once, to initialize an array of 4-byte fields. No masks are necessary. Later, in a produc-

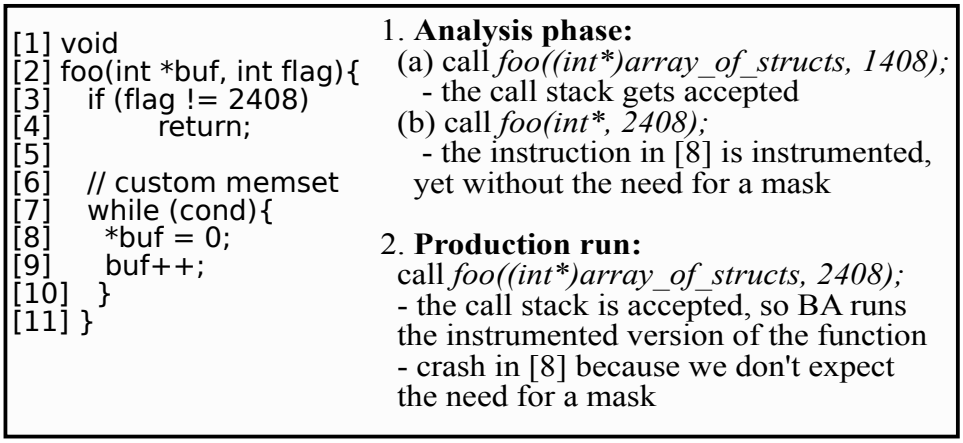


Fig. 5.3: BA-fields mode: a scenario leading to false positives.

tion run, the program takes a previously unknown code path, and uses the same function to access an array of 16-byte structures. Since it did not assign masks to this function originally, *BinArmor* now raises a (false) alarm.

To prevent the false alarm, we keep two versions of each function in a program: a vanilla one, and an instrumented one which performs color checking. When the program calls a function, *BinArmor* checks whether the function call also occurred at the same point during the analysis by examining the call stack. (In practice, we take the top 3 function calls.) Then, it decides whether or not to execute the instrumented version of the function. It performs color checking only for layouts of data structures we *have seen before*, so we do not encounter code that accesses memory in an unexpected way.

5.6.4 Are False Positives Still Possible?

While the extra mechanism to prevent false positives based on context checks is effective in practice, it does not give any strong guarantees. The problem is that a call stack does not identify the execution context with absolute precision. Fig. 5.3 shows a possible problematic scenario. In this case, it should not be the call stack, but a node in the program control flow graph which identifies the context. Only if we saw the loop in lines [6-9] initializing the `array_of_structs`, should we allow for an instrumented version of it at runtime. Observe that the scenario is fairly improbable. First, the offensive function must exhibit the need for masks, that is, it must access subsequent memory locations through a pointer to a previous field. Second, it needs to be called twice with very particular sets of arguments before it can lead to the awkward situation.

As we did not encounter false positives in *any* of our experiments, and BA-fields mode offers powerful, fine-grained protection, we think that the risk may be acceptable in application domains that can handle rare crashes.

5.7 Efficient Implementation

Protection by color matching combined with masks for color blindness allows *BinArmor* to protect data structures at a finer granularity than previous approaches. Even so, the mechanisms are sufficiently simple to allow for efficient implementations. *BinArmor* is designed to instrument 32-bit ELF binaries for the Linux/x86 platforms. Like Pebil [170], it performs static instrumentation, i.e., it inserts additional code and data into an executable, and generates a new binary with permanent modifications. We first describe how *BinArmor* modifies the layout of a binary, and next present some details of the instrumentation. (For a full explanation refer to [233].)

5.7.1 Updated Layout of ELF Binary

To accommodate new code and data required by the instrumentation, *BinArmor* modifies the layout of an ELF binary. The original data segment stays unaltered, while we modify the text segment only in a minor way—just to allow for the selection of the appropriate version of a function (Section 5.6.3), and to assure that the resulting code works correctly—mainly by adjusting jump targets to reflect addresses in the updated code (refer to Section 5.7.2). To provide protection, *BinArmor* inserts a few additional segments in the binary: BA_Initialized_Data, BA_Uninitialized_Data, BA_Procedures, and BA_Code.

Both data segments–*BA_(Un)Initialized_Data*– store data structures that are internally used by *BinArmor*, e.g., data structures color maps, or arrays mapping addresses in the new version of a binary to the original ones. The *BA_Procedures* code segment contains various chunks of machine code used by instrumentation snippets (e.g., a procedure that compares the color of a pointer with the color of a memory location). Finally, the *BA_Code* segment is the pivot of the *BinArmor* protection mechanism—it contains the original program's functions instrumented to perform color checking.

5.7.2 Instrumentation Code

To harden the binary, we rewrite it to add instrumentation to those instructions that dereference the array pointers. In BA-fields mode, we use multi-shade colors only if the data structures are nested. When we can tell that a variable is a string, or some other non-nested array, we switch to a simpler, single-level color check.

To provide protection, *BinArmor* reorganizes code at the instruction level. We do not need to know function boundaries, as we instrument instructions which were classified as array accesses, along with pointer move or pointer initialization instructions, during the dynamic analysis phase. We briefly describe the main steps taken by *BinArmor*: (1) inserting trampolines and method selector, (2) code relocation, (3) inserting instrumentation.

Inserting trampolines and method selector. The role of a *method selector* is to decide whether a vanilla or an instrumented function should be executed (see Section 5.6.3), and then jump to it. In *BinArmor*, we place a *trampoline* at the beginning of each (dynamically detected) function in the original text segment, which jumps to the method selector. The selector picks the right code to continue execution, as discussed previously.

Code relocation. *BinArmor*'s instrumentation framework must be able to add an arbitrary amount of extra code between any two instructions. In turn, targets of *all* `jump` and `call` instructions in a binary need to be adjusted to reflect new values of the corresponding addresses. As far as direct/relative jumps are concerned, we simply calculate new target addresses, and modify the instructions. Our solution to indirect jumps is similar to [232]: they are resolved at runtime, by using arrays maintaining a mapping between old and new addresses.

Inserting instrumentation. Snippets come in many shapes. For instance, snippets to handle pointer dereferences, to handle pointer move instructions, or to color memory returned by `malloc`. Some instructions require that a snippet performs more than one action. For example, an instruction which stores a pointer into an array, needs to both store the color of the pointer in a color map, and make sure that the store operation is legal. For an efficient instrumentation, we have developed a large number of carefully tailored snippets.

Colors map naturally on a sequence of small numbers. For instance, each byte in a 32-bit or 64-bit word may represent a shade, for a maximum nesting level of 4 or 8, respectively. Doing so naively, incurs a substantial overhead in memory space, but, just like in paging, we need only allocate memory for color tags when needed. The same memory optimization is often used in dynamic taint analysis. In the implementation evaluated in Section 5.8, we use 32 bit colors with four shades and 16 bit tags.

Fig. 5.4 shows an example of an array dereference in the binary hardened by *BinArmor*. The code is simplified for brevity, but otherwise correct. We see that each array dereference incurs some extra instructions. If the colors do not match, the system crashes. Otherwise, the dereference executes as intended. We stress that the real implementation is more efficient. For instance, adding two `call` instructions would be extremely expensive. In reality, *BinArmor* uses code snippets tailored to performance

5.8 Evaluation

We evaluate *BinArmor* on performance and on effectiveness in stopping attacks. As the analysis engine is based on the Qemu processor emulation, which is currently only available on Linux, all examples are Linux-based. However, the approach is not specific to any operating system.

```
_dereference_check_start:
# check whether tag value matches the pointer
cmp %edx, register_tag_edx
jne _dereference_check_end
[save %eax and %ebx used in instrumentation]
lea (%edx, %eax, 4), %ebx
call get_color_of_ebx ; loaded to %bx
mov register_color_edx, %ax
call color_match ; compare colors in %ax and %bx
cmpl $0, %eax    ; check result
je _dereference_ok

_dereference_bad:
["crash"]

_dereference_ok:
[restore %eax and %ebx used in instrumentation]

_dereference_check_end:
movl $0x1234, (%edx, %eax, 4); execute original instr
```

Fig. 5.4: Instrumentation for an array pointer dereference (with 16b colors and tags). The original instruction is `mov 0x1234,(%edx,%eax,4)`. We replace it by code similar to that presented in the figure (but more efficient).

We have performed our analysis for binaries compiled with two compiler versions, `gcc-3.4` and `gcc-4.4`, and with different optimization levels. All results presented in this section are for binaries compiled with `gcc-4.4 -O2` and without symbols, i.e., completely stripped. We reconstruct the symbols using Howard [227].

Performance To evaluate the performance of *BinArmor* operating in BA-fields mode[4], we compare the speed of instrumented (armored) binaries with that of unmodified implementations. Our test platform is a Linux 2.6 system with an Intel(R) Core(TM)2 Duo CPU clocked at 2.4GHz with 3072KB L2 cache. The system has 4GB of memory. For our experiments we used an Ubuntu 10.10 install. We ran each test multiple times and present the median. Across all experiments, the 90th percentiles were typically within 10% and never more than 20% off the mean.

We evaluate the performance of *BinArmor* with a variety of applications—all of the well-known *nbench* integer benchmarks [4]—and a range of real-world programs. We picked the *nbench* benchmark suite, because it is compute-intensive and several of the tests should represent close to worst-case scenarios for *BinArmor*.

For the real-world applications, we chose a variety of very different programs: a network server (`lighttpd`), several network clients (`wget`, `htget`), and a more compute-intensive task (`gzip`). `Lighttpd` is a high-performance web server used by such popular sites as YouTube, SourceForge, Wikimedia, Meebo, and ThePirateBay. `Wget` and `htget` are well-known command-line web clients. `Gzip` implements the DEFLATE algorithm which includes many array and integer operations.

Fig. 5.5 shows that for real I/O-intensive client-side applications like `wget` and `htget` the slowdown is negligible, while `gzip` incurs a slow-down of approximately 1.7x. As `gzip` is a very expensive test for *BinArmor*, the slow-down was less than we expected. The overhead for a production-grade web server like `lighttpd` is also low: less than 2.8x for all object sizes, and as little as 16% for large objects. In networking applications I/O dominates the performance and the overhead of *BinArmor* is less important.

Fig. 5.6 shows the results for the very compute-intensive *nbench* test suite. The overall slowdown for nbench is 2.9x. Since this benchmark suite was chosen as worst-case scenario and we have not yet fully optimized *BinArmor*, these results are quite good. Some of the tests incurred a fairly minimal slow-down. Presumably, these benchmarks are dominated by operations other than array accesses. String sort and integer sort, on the other hand, manipulate strings and arrays constantly and thus incur much higher slowdowns. They really represent the worst cases for *BinArmor*.

Effectiveness Table 5.1 shows the effectiveness of *BinArmor* in detecting attacks on a range of real-life software vulnerabilities. Specifically, these attacks represent

[4]The reason is that BA-fields mode is the most fine-grained, although in practice, the performance of BA-objects mode is very similar.

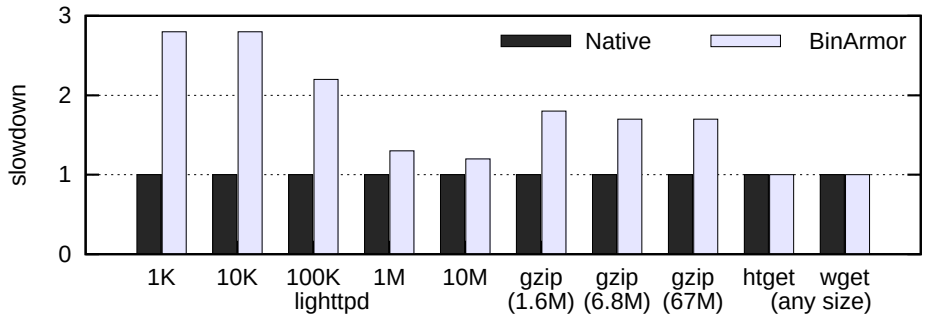


Fig. 5.5: Performance overhead for real world applications: *lighttpd* – for 5 object sizes (in connections/s as measured by *httperf*), *gzip* – for 3 object sizes, *htget* and *wget*.

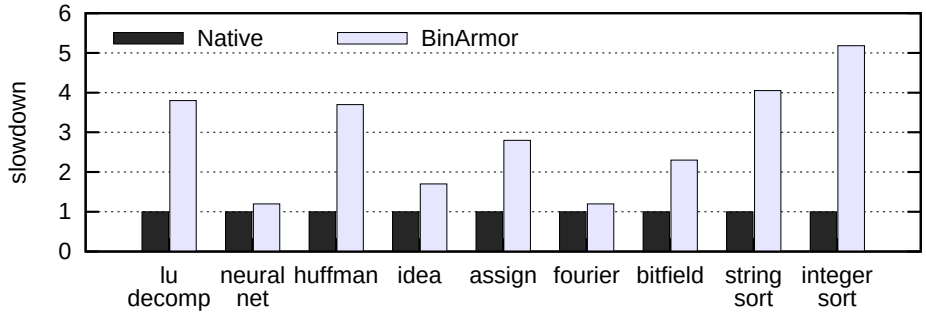


Fig. 5.6: Performance overhead for the compute-intensive *nbench* benchmark suite.

all vulnerabilities on Linux programs for which we found working exploits. *BinArmor* operating in either mode detected all attacks we tried and did not generate any false positives during any of our experiments. The attacks detected vary in nature and include overflows on both heap and stack, local and remote, and of both control and non-control data.

The detection of attacks on non-control data is especially encouraging. While control flow diversions would trigger alerts also on taint analysis systems like Argos [208] and Minemu [49], to the best of our knowledge, no other security measure for stripped binaries would be able to detect such attacks. As mentioned earlier, security experts expect non-control data attacks to become even more important attack vectors in the near future [67, 230].

Application	Vulnerability type	Security advisory
Aeon 0.2a	Stack overflow	CVE-2005-1019
Aspell 0.50.5	Stack overflow	CVE-2004-0548
Htget 0.93 (1)	Stack overflow	CVE-2004-0852
Htget 0.93 (2)	Stack overflow	
Iwconfig v.26	Stack overflow	CVE-2003-0947
Ncompress 4.2.4	Stack overflow	CVE-2001-1413
Proftpd 1.3.3a	Stack overflow	CVE-2010-4221
bc-1.06 (1)	Heap overflow	Bugbench [176]
bc-1.06 (2)	Heap overflow	Bugbench [176]
Exim 4.41	Heap overflow*	CVE-2010-4344
Nullhttpd-0.5.1	Heap overflow†	CVE-2002-1496
Squid-2.3	Heap overflow†	Bugbench [176]

* A non-control-diverting attack. † A reproduced attack.

Table 5.1: Tested vulnerabilities: *all attacks were stopped by BinArmor*, including the attack on non-control data.

5.9 Related Work

The easiest way to prevent memory corruptions is to do so at the source level, using a safe language or compiler extension. However, as access to source code or recompilation is often not an option, many binaries are left unprotected. Our work is about protecting binaries. Since it was inspired by the WIT compiler extension, we briefly look at compile time solutions also.

Protection at compile time Managed languages like Java and C# are safe from buffer overflows by design. Cyclone [149] and CCured [74] show that similar protection also fits dialects of C—although the overhead is not always negligible. Better still, data flow integrity (DFI) [60], write integrity testing (WIT) [22], and baggy

bounds checking (BBC) [23] are powerful protection approaches against memory corruptions for unmodified C.

BinArmor was inspired by the WIT compiler extension—a defense framework that marries immediate (fail-stop) detection of memory corruption to excellent performance. WIT assigns a color to each object in memory and to each write instruction in the program, so that the color of memory always matches the color of an instruction writing it. Thus all buffers which can be potentially accessed by the same instruction share the same color. WIT employs points-to analysis to find the set of objects written by each instruction. If several objects share the same color, WIT might fail to detect attacks that use a pointer to one object to write to the other. To get a grasp of the precision, we implemented points-to analysis ourselves, and applied it to global arrays in `gzip-1.4`. Out of 270 buffers, 124 have a unique color, and there are two big sets of objects that need to share it: containing 64 and 68 elements. (We assume that we provide templates for the `libc` functions. Otherwise, the precision is worse.) To deal with these problems, WIT additionally inserts small guards between objects, which cannot be written by any instruction. They provide an extra protection against sequential buffer overflows. *BinArmor* tracks colors of objects dynamically, so each array is assigned a unique color.

Also, WIT and BBC protect at the granularity of memory allocations. If a program allocates a structure that contains an array as well as other fields, overflows within the structure go unnoticed. As a result, the attack surface for memory attacks is still huge. SoftBound is one of the first tools to protect subfields in C structures [197]. Again, SoftBound requires access to source code.

BinArmor's protection resembles that of WIT, but without requiring source code, debugging information, or even symbol tables. Unlike WIT, it protects at the granularity of subfields in C `structs`. It prevents not just out-of-bounds writes, as WIT does, but also reads. As a drawback, *BinArmor* may be less accurate, since dynamic analysis may not cover the entire program.

Protection of binaries Arguably some of the most popular measures to protect against memory corruption are memory debuggers like Purify and Valgrind [198]. These powerful testing tools are capable of finding many memory errors without source code. However, they incur overheads of an order of magnitude or more. Moreover, their accuracy depends largely on the presence of debug information and symbol tables. In contrast, *BinArmor* is much faster and requires neither.

One of the most advanced approaches to binary protection is XFI [108]. Like memory debuggers, XFI requires symbol tables. Unlike memory debuggers, DTA, or *BinArmor*, XFI's main purpose is to protect host software that loads modules (drivers in the kernel, OS processes, or browser modules) and it requires explicit support from the hosting software–to grant the modules access to a slice of the address space. It offers protection by a combination of control flow integrity, stack splitting, and memory access guards. Memory protection is at the granularity of

the module, and for some instructions, the function frame. The memory guards will miss most overflows that modify non-control data.

An important class of approaches to detect the *effects* of memory corruption attacks is based on dynamic taint analysis (DTA) [75]. DTA does not detect the memory corruption itself, but may detect malicious control flow transfers. Unfortunately, the control flow transfer occurs at a (often much) later stage. With typical slowdowns of an order of magnitude, DTA in software is also simply *too expensive* for production systems.

Non-control data attacks are much harder to stop [67]. [66] pioneered an interesting form of DTA to detect some of these attacks: pointers become tainted if their values are influenced by user input, and an alert is raised if a tainted value is dereferenced. However, pointer taintedness for detecting non-control data attacks is shown to be impractical for complex architectures like the x86 and popular operating systems [226]. The problems range from handling table lookups to implicit flows and result in false positives and negatives. Moreover, by definition, pointer taintedness cannot detect attacks that do not dereference a tainted pointer, such as an attack that would overwrite the `privileged` field in Fig. (5.2a).

5.10 Discussion

Obviously, *BinArmor* is not flawless. In this section, we discuss some generic limitations.

With a dynamic approach, *BinArmor* protects only arrays detected by Howard. If the attackers overflow other arrays, we will miss the attacks. Also, if particular array accesses are not exercised in the analysis phase, the corresponding instructions are not instrumented either. Combined with the tags (Section 5.5.2), this lack of accuracy can only cause false negatives, but never false positives. In practice, as we have seen in Section 5.8, *BinArmor* was able to protect all vulnerable programs we tried.

Howard itself is designed to err on the safe side. In case of doubt, it overestimates the size of an array. Again, this can lead to false negatives, but not false positives. However, if the code is strongly obfuscated or deliberately designed to confuse Howard, we do not guarantee that it will never misclassify a data structure in such a way that it will cause a false positive. Still, it is unlikely, because to do so, the behavior during analysis should also be significantly different from that during the production run. In our view, the risk is acceptable for software deployments that can tolerate rare crashes.

We have implemented two versions of *BinArmor*: BA-objects mode, and BA-fields mode. While the latter protects memory at a fine-grained granularity, there exist theoretical situations that can lead to false alerts. However, in practice we did not encounter any problems. Since the protection offered is very attractive — *BinArmor* protects individual fields within structures — we again think that the risk is acceptable.

Code coverage is a limitation of all dynamic analysis techniques and we do not claim any contribution to this field. Interestingly, code coverage can also be 'too good'. For instance, if we were to trigger a buffer overflow during the analysis run, *BinArmor* would interpret it as normal code behavior and not prevent similar overruns during production. Since coverage techniques to handle complex applications are currently still fledgling, this is mostly an academic problem. At any rate, if binary code coverage techniques are so good as to find such real problems in the testing phase, this can only be beneficial for the quality of software.

5.11 Future work

BinArmor's two main problems are accuracy (in terms of false negatives and false positives) and performance (in terms of the slowdown of the rewritten binary). In this section, we discuss ways to address these problems.

First, the root cause of *BinArmor*'s false negative and false positive problems is the lack of code coverage. Our next target, therefore, is to extend the paths covered dynamically by means of static analysis. For instance, we can statically analyze the full control flow graphs of all functions called at runtime. Static analysis in general is quite hard, due to indirect calls and jumps, but within a single function indirect jumps are often tractable (they are typically the result of `switch` statements that are relatively easy to handle).

Second, the cause of *BinArmor*'s slowdown is the instrumentation that adds overhead to every access to an array that *BinArmor* discovered. We can decrease the overhead using techniques that are similar to those applied in WIT. For instance, there is no need to perform checks on instructions which calculate the address to be dereferenced in a deterministic way, say at offset 0x10 from a base pointer. Thus, our next step is to analyze the instructions that are candidates for instrumentation and determine whether the instrumentation is strictly needed.

5.12 Conclusions

We described a novel approach to harden binary software proactively against buffer overflows, without access to source or symbol tables. Besides attacks that divert the control flow, we also detect attacks against non-control data. Further, we demonstrated that our approach stops a variety of real exploits. Finally, as long as we are conservative in classifying data structures in the binaries, our method will not have false positives. On the downside, the overhead of our approach in its current form is quite high—making it unsuitable for many application domains today. However, we also showed that significant performance optimizations may still be possible. It is our view that protection at the binary level is important for dealing with real threats to real and deployed information systems.

6

Online and Scalable Data Validation in Advanced Metering Infrastructures

Preamble: Relation to the Research Roadmap

As introduced by the Peccadillo scenario (roadmap deliverable D4.1) and further discussed in Chapter 2, detecting the possible tampering of embedded devices' firmware which could result in fraudulent energy consumption reports is of fundamental importance.

In a broader sense, the detection of all fraudulent and incorrect data, caused either by malicious users or by the faulty and lossy nature of embedded devices, is key for the proper functioning of the applications that leverage such data.

In this chapter, we discuss how data can be validated in an on-line and scalable fashion meeting the high-throughput and low-latency requirements of Advanced Metering Infrastructures. At the same time, we show how user-defined validation rules can be easily composed in order to tune the analysis to the hardware (e.g., the embedded devices) producing the data.

Vincenzo Gulisano, Magnus Almgren, Marina Papatriantafilou; "Online and Scalable Data Validation in Advanced Metering Infrastructures" IEEE PES Innovative Smart Grid Technologies (ISGT) European Conference - October 2014

Abstract

The shift from traditional to cyber-physical grids involves the deployment of Advanced Metering Infrastructures, networks of communication-enabled devices remotely controlled by utilities. Live information collected by these devices enables for applications such as demand/response, real-time pricing or intrusion detection, among others. In these scenarios, data validation is necessary in order to preprocess the noisy and lossy data produced by the devices and make it available to utilities' or third parties' applications. Challenges proper of data validation in this domain include the possibility of expressing validation rules specific to an Advanced Metering Infrastructure installation and analysis techniques that cope with the large and fluctuating volume of data produced by the devices.

In this paper, we discuss and provide evidence of the online, scalable and expressive validation analysis enabled by the data streaming processing paradigm. Based on a prototype implementation on top of the Storm processing engine and using data from a real-world Advanced Metering Infrastructure, we show that streaming-based validation rules enable for the analysis of thousands of meters per second and only incur in small latency penalties in the order of milliseconds.

6.1 Introduction

The transition from traditional to cyber-physical electric grids involves the deployment of Advanced Metering Infrastructures (AMIs). An AMI is composed by networks of communication-enabled devices that share information (e.g., energy consumption readings, energy quality measurements or power outage logs) with the utility's head-end. A considerable number of research directions surrounds AMIs: real-time pricing [24], demand-response [130], users' privacy [245], smart meters vulnerabilities [76], Intrusion Detection Systems (IDSs) [39], users' awareness and social media [185] or load forecast [118]. All these research fields depend on the data produced by AMIs' devices. Unfortunately, such data is known to be noisy, lossy and to be possibly delivered out of order and with duplicates (especially for AMIs relying on wireless communication [158]). Because of this, data validation analysis is adopted to preprocess and clean the data collected from the devices before the utility or third parties access it. Such validation analysis usually relies on a set of *validation rules*. It should be noted that noisy and missing data does not depend uniquely on AMIs' devices themselves, but can also be caused by their (possibly malicious) users. Causes of noisy and lossy data include faulty or badly calibrated devices, lossy or overloaded (or possibly jammed) communication channels or incorrect energy consumption readings manipulated by malicious users, among others.

Challenges Millions of messages are generated on a daily basis by AMIs' devices. Some of them are generated with a certain periodicity chosen by the utility

(e.g., energy consumption readings) while others can appear in bursts over time (e.g., power outage logs). To this end, a scalable validation analysis is required in order to cope with the large and fluctuating volume of data produced by the devices. At the same time, an online (i.e., in a real-time fashion) validation analysis is required for live information to be leveraged in scenarios such as real-time pricing or defense frameworks. Since AMIs are composed by heterogeneous types and brands of devices (e.g., electricity meters, meter concentrator units, heating meters, and so on) that usually rely on proprietary data formats, it is desirable for utilities to rely on validation tools with which system experts can easily compose validation rules rather than rely on a set of predefined ones.

Contributions In this paper, we propose and provide evidence about the online, scalable and expressive validation analysis enabled by the data streaming processing paradigm. The latter has been proposed as an alternative to the traditional "store-then-process" (database) paradigm by applications demanding for high processing capacity with low processing latency guarantees (e.g., financial markets analysis [235], publicity pricing [25], fraud detection [129] or defense frameworks [56]). In data streaming, *continuous queries* are defined as Directed Acyclic Graphs (DAGs) of operators and run in a distributed and parallel fashion by Stream Processing Engines (SPEs) such as Storm [236], Yahoo S4 [259] or StreamBase [238]. As we will discuss, *streaming-based* validation rules can be composed by means of the standard data streaming operators provided by these SPEs. We provide the following contributions:

1. An analysis of the expressive and scalable validation enabled by the data streaming processing paradigm, including examples of real-world validation rules.

2. An implementation of a set of streaming-based validation rules on top of Storm [236], a state of the art SPE used in companies such as Twitter.

3. An evaluation of the performance and scalability of such streaming-based validation rules based on data extracted from a real-world AMI.

The rest of the paper is organized as follows. We introduce some preliminary concepts in Section 6.2. We present how data streaming operators can be used to compose streaming-based validation rules in Section 6.3. We provide an evaluation of the performance (in terms of throughput and latency) of a set of validation rules in Section 6.4. Section 6.5 discussed the related work while Section 6.6 concludes the paper.

6.2 System Model

6.2.1 Validation Analysis in Advanced Metering Infrastructures

AMIs networks are composed by heterogeneous devices such as smart meters (in charge of forwarding readings about electricity, gas or water consumption) and meter concentrators units (in charge of collecting consumption readings to forward them to the utility head-end). These devices are usually resource-constrained and organized in different network topologies (e.g., point-to-point, hierarchical or mesh ones). For these reasons, it is more desirable for a utility to be able to compose validation rules rather than rely on a predefined set of validation rules that might not fit local constraints (or rely on ad hoc solutions that will be hard to maintain). To this end, data streaming operators constitute excellent building blocks to compose streaming-based validation rules, as we discuss in Section 6.3.

The large and fluctuating volume of data produced by AMIs' devices demand for a scalable and online validation analysis in order for the data to be leveraged by applications such as real-time pricing or defense frameworks. Validation rules could be deployed in different ways within an AMI. On one hand, they could be deployed at the utility head-end system. This centralized approach is the common choice in existing AMIs (e.g., for IDSs, as discussed in [127]). Data collected from the devices is preprocessed as it enters the utility's head-end and subsequently stored in order to be available to other applications. On the other hand, the deployment could push the validation analysis to the meters themselves (e.g., discarding wrong consumption readings at the meters rather than at the head-end). As discussed in the context of COUGAR [47], one of the pioneer SPEs, the performance of a given traffic analysis technique can be improved by keeping the data moved across devices to its minimum (i.e., transferring only useful information). Intermediate solutions could rely both on the devices themselves and the utility's head-end to perform the validation analysis. To our advantage, SPEs would allow for streaming-based validation rules to be executed at arbitrary numbers of heterogeneous nodes, thus leveraging any of these possible deployment strategies, as we discuss later in Section 6.3.3.

6.2.2 Meter Data Management systems and Validation, Estimation and Editing (VEE) Rules

The systems in charge of collecting and storing AMIs' data are referred to as Meter Data Management systems [128, 204]. Such systems rely on a set of Validation, Estimation and Editing (VEE) rules that are used to preprocess the information before the latter is stored in the utility's databases.

In the context of VEE rules, *Validation* refers to the purging of noisy or possibly corrupted information. As discussed in Section 6.1, possible causes of such information degradation include faulty devices or overloaded communication channels, among others. Possible examples of validation include removal of negative

consumption values or consumption values that exceed the capacity of the fuse installed in a smart meter.

Estimation refers to the ability of producing missing information. Such information could be produced by relying on interpolation methods or by methods that take into account meters' historical data. As an example, a period of missing consumption values for a given smart meter could be estimated based on its consumption as observed during the previous six months.

Editing refers to the ability of modifying historical information. The rationale is that, in scenarios such as real-time pricing, live information (e.g., energy consumption readings) is needed within specified time intervals. To comply with such time constraints, it might be preferable for the utility to rely on estimated data rather than postponing any computation until all data is available. Nevertheless, real information that arrives at the utility after having been estimated (e.g., because of out of order delivery of messages) is still more appropriate to store than the calculated estimate.

6.2.3 Data Streaming

In data streaming, incoming data is processed by means of *continuous queries* (or simply queries), DAGs where vertices represent operators and edges specify how tuples flow between them [19]. Differently from their database counterpart, such queries are not issued at a point in time but rather stand continuously to process information on the fly, updating their computation and producing results accordingly. Data streaming queries consume streams, each defined as an unbounded sequence of tuples sharing the same schema, composed by attributes $\langle ts, A_1, A_2, \ldots, A_n \rangle$. Given a tuple t, attribute $t.ts$ represents its creation timestamp at the data source while $\langle A_1, A_2, \ldots, A_n \rangle$ are application related attributes. Figure 6.1 presents a sample schema for tuples referring to energy consumption readings that could be produced by the smart meters deployed in an AMI. In the example, tuples are composed by attributes *ts*, the creation timestamp, *sm* the smart meter id, and *cons*, the consumption in kWh observed during the last hour.

Attribute	ts	sm	cons
Description	Creation timestamp	Meter ID	Hourly consumption (kWh)

Fig. 6.1: Sample schema of tuples carrying energy information readings.

Data streaming operators are distinguished into *stateless* and *stateful*. Stateless operators (e.g., Filter, Map, Union) process each tuple individually. On the other hand, stateful operators (e.g., Aggregate, Join) perform computations based on sequences of tuples. Due to the unbounded nature of streams, stateful computations are performed over *sliding windows* (or simply windows). Windows can be *time-based* (e.g., the tuples received in the last 10 minutes) or *tuple-based* (e.g., the last 20 received tuples), and are defined by parameters *Size* and *Advance*, specifying the

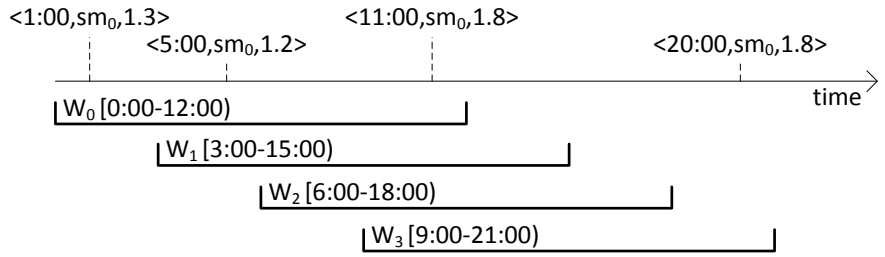


Fig. 6.2: Sample sequence of tuples carrying energy information readings and evolution of a time-based window of *Size* and *Advance* of 12 and 3 hours, respectively.

extent of a window and the amount of information discarded each time the latter slides.

Figure 6.2 present a sample sequence of tuples composed by the attributes shown in Figure 6.1 and the evolution of a time-based window of size and advance of 12 and 3 hours, respectively.

6.3 Streaming-based validation analysis

This section discusses how data streaming can be used to run data validation analysis in AMIs. We first provide definitions and examples of a basic set of data streaming operators and continue discussing how such operators can be used to compose streaming-based validation rules. We conclude discussing how such rules are executed by SPEs at an arbitrary number of nodes, thus leveraging the possible deployment options discussed in Section 6.2.1.

6.3.1 Basic data streaming operators

As discussed in Section 6.2.1, it is desirable for a utility to compose validation rules specific to their AMI installation rather than using a predefined set of validation rules. By relying on the data streaming paradigm, this would be done by composing queries using the set of operators made available by a given SPE. Several relational data streaming operators are usually provided by SPEs (including Filter, Map, Union, Aggregate, Join and Sort [18]). To give evidence of the expressiveness of data streaming queries, we discuss in the following how four of these operators, usually defined by all SPEs, allow to compose validation rules: Filter, Map, Aggregate and Join. For each of these operators, we provide its definition and describe its semantic together with an example (all the examples refer to the sample schema shown in Figure 6.1). Each operator is defined as:

$$OP\{P_1, \ldots, P_m\}(I_1, \ldots, I_n, O_1, \ldots, O_p)$$

where OP represents the operator name, $P_1, \ldots, P_m$ represent a set of parameters that specify the operator semantics (e.g., functions used to transform input tuples,

predicates used to decide which information to discard or parameters related to the windowing model), $I_1, \ldots, I_n$ a set of input streams and $O_1, \ldots, O_p$ a set of output streams. Optional parameters are defined using square brackets.

Filter

This stateless operator is a generalized selection operator that can be used to discard tuples. The operator is defined as:

$$F\{C\}(I, O)$$

where I is the input stream, O is the output stream and C is the filtering condition. Each incoming tuple is forwarded to O if the filtering condition C holds. The operator does not modify the schema of its input tuples.

The following example considers a Filter operator forwarding only input tuples referring to a consumption reading lower than or equal to 10 kWh.

$$F\{\text{cons} \leq 10\text{kWh}\}(I, O)$$

Map

This stateless operator is a generalized projection operator that can be used to transform one input tuple into one or multiple output tuples with a different schema. The operator is defined as:

$$M\{A'_1 \leftarrow f_1(t_{in}), \ldots, A'_n \leftarrow f_n(t_{in})\}(I, O)$$

where I and O represent the input and output streams, respectively. t_{in} is a generic input tuple and the attributes $\langle A'_1, \ldots, A'_n \rangle$ form the output tuples' schema.

The following example considers a Map operator converting the attribute *cons* of each input tuple from kWh to Wh.

$$M\{\text{ts} \leftarrow \text{ts}, \text{sm} \leftarrow \text{sm}, \text{cons} \leftarrow \text{cons}/1000\}(I, O)$$

Aggregate

This stateful operator is used to compute aggregation functions such $mean$, $count$, min, max, $first$ or $last$ over windows of tuples. The operator is defined as:

$$\begin{aligned} A\{&\text{WType}, \text{Size}, \text{Advance}, A'_1 \leftarrow f_1(W), \ldots, A'_n \leftarrow f_n(W) \\ &[, \text{Group-by}=(A_{i_1}, \ldots, A_{i_m})]\}(I, O) \end{aligned}$$

$WType$ specifies the window type: $time$ for time-based windows or $tuples$ for tuple-based ones. Parameters *Size* and *Advance* specify the amount of tuples to be maintained and discarded each time the window slides. An output tuple carrying the result of functions $f_1(W), \ldots, f_n(W)$ is produced each time the window

slides if the latter contains at least one tuple. If parameter *Group-by* is set, a separate window will be maintained for each distinct value of attributes $A_{i_1}, \ldots, A_{i_m}$. Output tuples' schema is composed by attributes ts (the timestamp of the window), *Group-by* (if defined) and attributes $A'_1, \ldots, A'_n$.

The following example considers an Aggregate operator used to compute the highest consumption reading of each smart meter over a window of 24 hours, producing a result every hour.

$$A\{time, 24\text{h}, 1\text{h}, \text{max} \leftarrow max(\text{cons}), \text{Group-by=(sm)}\}(I, O)$$

In the example, $WType = time$. Window $Size$ and $Advance$ parameters are set to 24 hours and 1 hour, respectively. That is, an output tuple will be produced every hour, and will contain the highest consumption reading observed during the last 24 hours for each meter separately. By dropping the *Group-by* attribute, we would instead get the highest consumption reading among all meters. The schema associated to the output stream is composed by attributes $\langle$ts, sm, max$\rangle$.

Join

This stateful operator is used to match tuples from two distinct input streams, referred to as left (l) and right (r). An output tuple whose schema is the combination of the l and the r schema is produced each time a predicate P holds for a pair of tuples from the l and r streams. Similarly to the Aggregate operator, the Join uses windows to maintain input tuples. A separate window (time or tuple based) is maintained for the l and r streams. The operator is defined as:

$$J\{P, WType, Size\}(S_l, S_r, O)$$

The following example considers a Join operator that matches two tuples if they refer to the same energy consumption values and are observed during the last hour.

$$J\{l.cons = r.cons, time, 1h\}(S_l, S_r, O)$$

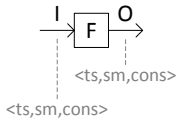
Since the operator combines the schema of each pair of l and r tuples that match the predicate, the output tuples will be composed by attributes $\langle$l.ts, l.sm, l.cons, r.ts, r.sm, r.cons$\rangle$

6.3.2 Composing streaming-based validation queries

As discussed in Section 6.2.2, data validation rules are used both for removing incoming data that should not be persisted and to estimate missing one. We present here sample queries that can be used to perform these two tasks.

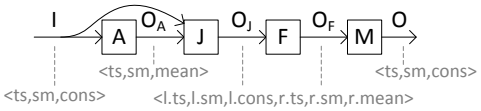
When composing a validation rule that prevents incorrect data from being stored, a Filter operator can be used to take decisions about which input tuples to discard. Such a decision can be taken in different ways. On one hand, the decision could be taken based only on the tuple itself. As presented in the sample Query 6.1 (validation rule V_1) this can be done with a single Filter operator (F).

Table 6.1: Validation rule V_1. Discard energy consumption readings referring to negative consumption values or consumption values exceeding a given threshold, here set to 10 kWh.



$F\{\text{cons} > 0 \wedge \text{cons} \leq 10\text{kWh}\}(I, O)$

Table 6.2: Validation rule V_2. Discard a meter's energy consumption readings if they exceed two times the mean consumption observed for the same meter during a given period, here set to the previous three hours.

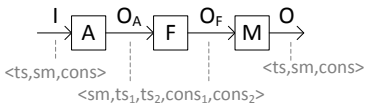


$A\{\text{time,3h,1h,mean=}mean\text{(cons), Group-by=sm}\}(I, O_A)$
$J\{\text{time,3h,l.sm=r.sm} \wedge \text{l.ts=r.ts+3h}\}(I, O_A, O_J)$
$F\{\text{l.cons} \leq \text{r.mean} \times 2\}(O_J, O_F)$
$M\{\text{ts} \leftarrow \text{l.ts}, \text{sm} \leftarrow \text{l.sm}, \text{cons} \leftarrow \text{l.cons}\}(O_F, O)$

On the other hand, the decision about which tuples to discard could be based on historical information. In general, an Aggregate, a Join and a Filter operator can be used to filter tuples based on historical information. The Aggregate operator can be used to compute the reference information used to decide which tuples to discard. Subsequently, the Join operator can be used to match each incoming tuple with its reference information. Finally, the Filter operator can be used to compare the incoming tuple and the reference information. Query 6.2 (validation rule V_2) presents a sample query that discards a tuple if its energy consumption exceeds more than two times the average observed for the same meter during the last three hours. Since the schema of the input tuples is modified by the Aggregate (A) and the Join (J) operators, a final Map operator (M) is used to convert the schema of the Filter's output tuples back to the same as the input ones.

When composing a validation rule that estimates missing information, the first task is to spot that one or multiple consumption readings are missing. This can be done by relying on an Aggregate and a Filter operator. The Aggregate operator

Table 6.3: Validation rule V_3. Interpolate missing consumption values if the time distance between two consecutive tuples from the same meter exceeds a given threshold, here set to one hour.



$A\{$tuples, 2, 1, ts_1=$first$(ts), ts_2=$last$(ts), $cons_1$= ...
... $first$(cons),$cons_2$=$last$(cons), Group-by=sm$\}(I, O_A)$
$F\{ts_2\text{-}ts_1 > 1h\}(O_A, O_F)$
$M\{$(ts $\leftarrow$ $interp$(ts_1,ts_2, sm $\leftarrow$ sm, ...
... cons $\leftarrow$ $interp$($cons_1$,$cons_2$)$\}(O_F, O)$

can match any pair of consecutive tuples referring to the same smart meter. Subsequently, the Filter can by used to check if their distance in time exceeds a given threshold.

Query 6.3 presents a validation rule (V_3) that interpolates missing consumption values if the time distance between two consecutive tuples from the same meter exceeds one hour. The query is composed by three operators. The Aggregate operator (A) defines a tuple-based window of size and advance of 2 tuples and 1 tuple, respectively. A tuple containing both the first and last timestamps (ts_1 and ts_2) and consumption values ($cons_1$ and $cons_2$) is created for each smart meter (*Group-by*=sm). Subsequently, the Filter operator (F) is used to forward only such tuples whose distance between ts_1 and ts_2 exceeds one hour. Finally, the Map operator (M) is used to interpolate missing values applying the function $interp$ (which would be defined by the system expert), which will create an output tuple composed by attributes $\langle$ts, sm, cons$\rangle$ for each missing hour. An Aggregate operator could maintain historical information (similarly to Query 6.2) to be used when estimating missing values (not shown here).

6.3.3 Scalable execution of streaming-based validation queries

A major advantage from the utility perspective is that SPEs can leverage the resources devoted to the validation analysis (e.g., nodes in a cluster or cores in a parallel architecture) independently of their deployment, as discussed in Section 6.2.1. In this section, we further discuss how SPEs allow for such scalable execution of streaming-based validation rules.

Scalable execution is achieved by means of distributed and parallel operator execution. Distributed execution (achieved by means of *inter-operator parallelism*) allows for operators belonging to the same query to be run at different nodes. At the same time, parallel execution (achieved by means of *intra-operator parallelism*) allows for individual operators to be run in parallel at arbitrary numbers of nodes.

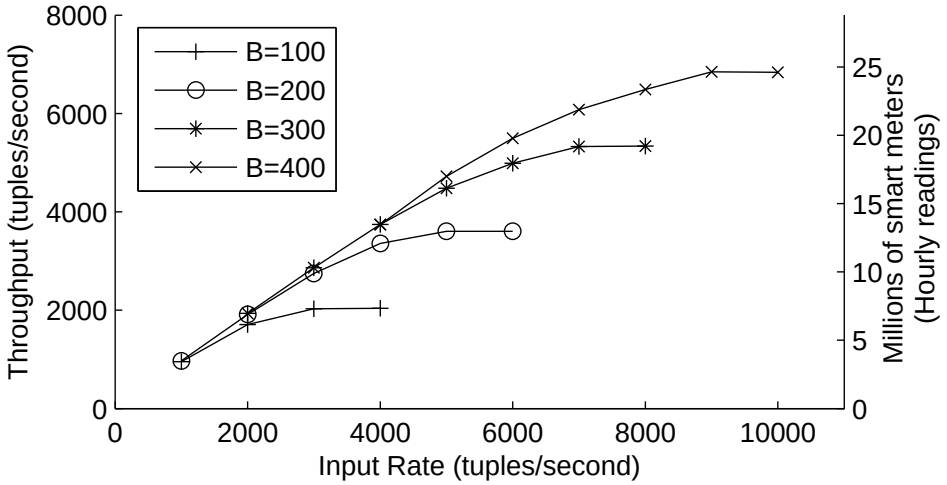


Fig. 6.3: Throughput (tuples/second) of validation rules V_1, V_2 and V_3 for different batch sizes.

Each operator is parallelized by deploying multiple instances of it and by partitioning its input stream(s) to such instances. The way in which the input tuples are routed to the different instances depends on whether the parallel operator is stateless or stateful. Since stateless operators (e.g., Filter and Map) process each tuple individually, tuples can be routed to the different instances by any routing technique (e.g., round-robin). Nevertheless, the routing of tuples feeding a stateful operator must be aware of its semantics. As an example, when deploying multiple instances of the Aggregate and Join operators of Query 6.2, all the tuples referring to the same smart meter must be routed to the same operator instance in order for the latter to produce the correct output tuple. We refer the reader to [129] for a exhaustive discussion about the parallelization of data streaming operators.

6.4 Evaluation

In this section, we evaluate the applicability of the data streaming paradigm in the context of AMIs' data validation. We first provide details about the data we use and the evaluation setup. Subsequently, we evaluate the performance in terms of *throughput* and *latency* of the streaming-based validation rules discussed in Section 6.3.

Evaluation setup

The real-world AMI used in our evaluation is composed of 300,000 smart meters that cover a metropolitan area of 450 km^2 with roughly 600,000 inhabitants. The utility extracted 13 months (From May 2012 to June 2013) of hourly consumption readings from a subset of 50 meters and made this dataset available for us. The

validation rules are implemented on top of Storm, version 0.9.1. The evaluation has been conducted with an Intel-based workstation with two sockets of 8-core Xeon E5-2650 processors and 64 GB DDR3 memory. All experiments start (resp. end) with a warm-up (resp. cool-down) phase. Presented results are measured in between of these phases and are averaged over 10 separate runs. In all experiments, we process the data extracted from the AMI, modifying only the rate at which tuples are injected.

Performance evaluation

In order to make energy consumption readings available in a real-time fashion, it is important to reduce the time that goes from the measurement at the meter itself to the delivery at the utilities' or third parties' applications. Such delay depends on two main factors: the period with which energy consumption readings are pulled from or pushed by meters (plus the network latency) and the latency introduced by the data validation analysis. A trade-off exists between these two factors. On one hand, it is desirable to retrieve energy consumption readings frequently (if possible, as soon as the measurement is taken). On the other hand, it is common to batch tuples together in order for high-throughput systems to achieve better performance. In this experiment, we study the throughput and latency achieved for different batch sizes B. As it would be done in a real deployment consuming live information, all the three validation rules are executed at the same time. The throughput is measured as the rate (tuple/second) that each validation rule can sustain while latency is measured for each validation rule individually.

Figure 6.3 presents the processing throughput for different batch sizes of 100, 200, 300 and 400 tuples. For each batch size, the throughput initially grows linearly with the increasing input rate while it flattens down when reaching the maximum processing capacity. As expected, increasing the batch size results in higher processing throughput. For a batch size of 100 tuples, the server is able to process approximately $2,000$ messages per second. For a batch size of 400 tuples, the throughput grows to approximately $7,000$ messages per second. As shown in the figure (secondary Y axis), if meters measure energy consumption readings every hour, this processing capacity would enable us to validate almost 25 million meters.

Figure 6.4 presents the latency (in milliseconds), for each batch size B and validation rule (bars are plotted in logarithmic scale to better emphasize the differences). It can be noticed that different latencies are imposed by each validation rule. At the same time, an increasing latency is also observed when increasing the batch size B. The highest processing latency is imposed by validation rule V_2, since it contains both an Aggregate and a Join operator (stateful operators' computations are more expensive and thus incur higher latencies than stateless ones). However, the latency introduced by the analysis is negligible with the one introduced by transferring the data from the meters to the utility (in our AMI, this action takes a time in the order of seconds).

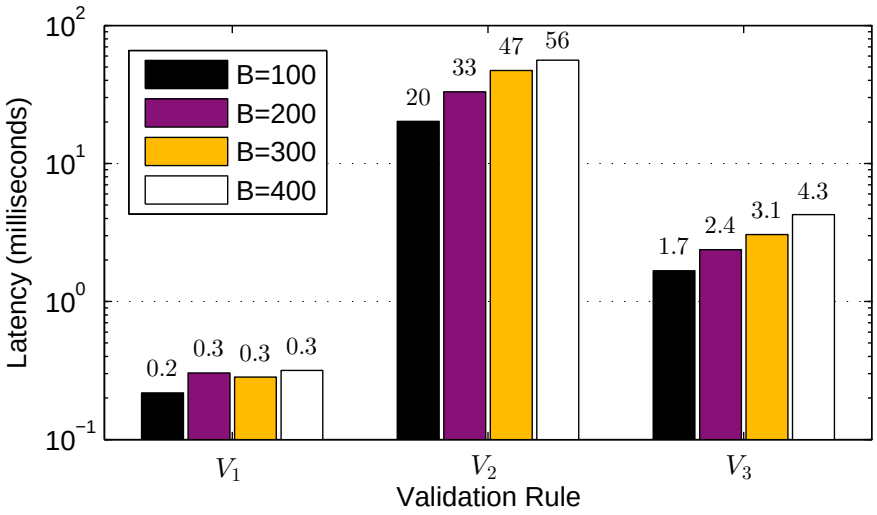


Fig. 6.4: Latency (milliseconds, logarithmic scale) of validation rules V_1, V_2 and V_3 for different batch sizes.

6.5 Related Work

The data streaming research field emerged around the year 2000 to overcome the limitations of traditional database approaches for data intensive applications such as fraud detection, IDS or financial market analysis. Sensor networks sharing requirements similar to AMIs' ones played a key role in the development of pioneer SPEs such as TelegraphCQ [63], Cougar [47] or Aurora [59]. During these years, data streaming research has focused on aspects such as processing of imprecise and missing information, graceful degradation and load shedding techniques under peaks of load, and Quality-of-Service (QoS) metrics.

To the best of our knowledge, this is the first work that focuses on the expressiveness and performance of data streaming queries in the context of AMIs' data validation. Nevertheless, the data streaming processing paradigm has been taken into account in other AMI-related scenarios, including electricity load forecast [118], IDSs [110] and cloud-infrastructures [175, 223]. These works provide evidence that the data streaming processing paradigm is an appropriate candidate to address the data analysis requirements proper of AMIs.

6.6 Conclusions

A considerable number of applications surrounding AMIs (e.g., demand-response, real time pricing and IDSs, among others) depend on the data produced by AMIs' devices. Its noisy and lossy nature demands for validation analysis in order to preprocess the data that is later accessed by utilities' or third parties' applications. In this paper, we have discussed how the data streaming processing paradigm can be leveraged to provide online and scalable validation analysis, composing valida-

tion rules by means of data streaming operators. Based on an implementation on top of the Storm SPE and conducted with data from a real-world AMI, we have shown that streaming-based validation rules allow for the analysis of thousands of energy consumption readings per second (with latencies in the realm of milliseconds) while relying on commodity hardware. Thanks to the distributed and parallel execution enabled by SPEs, streaming-based validation rules could also be run closer to the sources (e.g., achieving better scalability by filtering values at the sources and saving bandwidth).

7

METIS: a Two-Tier Intrusion Detection System for Advanced Metering Infrastructures

Preamble: Relation to the Research Roadmap

In our roadmap, we discuss "vital societal functions, health, safety, security, economic or social well-being of people" as one of the key aspects of cyber-physical systems (CPSs). Unfortunately, a wide spectrum of attacks can be launched in order to compromise them. A challenge in this context is the detection of malicious activities meant to go unnoticed (e.g., as the ones motivating the Peccadillo scenario, described in D4.1, and targeting domestic smart meters). It should be noticed that the absence of documented attacks and known signatures constitutes one of the challenges of such detection.

In this chapter, we propose a framework for the detection of such malicious activities. As discussed, the framework does not only ease the modeling of possible adversary goals but also addresses the high-throughput and low-latency requirements needed to process CPSs' data in an on-line fashion and detect threats timely.

Vincenzo Gulisano, Magnus Almgren, Marina Papatriantafilou; "METIS: a Two-Tier Intrusion Detection System for Advanced Metering Infrastructure" 10th International Conference on Security and Privacy in Communication Networks (SecureComm) - September 2014

Abstract

In the shift from traditional to cyber-physical electric grids, motivated by the needs for improved energy efficiency, Advanced Metering Infrastructures have a key role. However, together with the enabled possibilities, they imply an increased threat surface on the systems. Challenging aspects such as scalable traffic analysis, timely detection of malicious activity and intuitive ways of specifying detection mechanisms for possible adversary goals are among the core problems in this domain.

Aiming at addressing the above, we present *METIS*, a *two-tier streaming-based intrusion detection framework*. *METIS* relies on probabilistic models for detection and is designed to detect challenging attacks in which adversaries aim at being unnoticed. Thanks to its two-tier architecture, it eases the modeling of possible adversary goals and allows for a fully distributed and parallel traffic analysis through the data streaming processing paradigm. At the same time, it allows for complementary intrusion detection systems to be integrated in the framework.

We demonstrate *METIS*' use and functionality through an *energy exfiltration* use-case, in which an adversary aims at stealing energy information from AMI users. Based on a prototype implementation using the Storm Stream Processing Engine and a very large dataset from a real-world AMI, we show that *METIS* is not only able to detect such attacks, but that it can also handle large volumes of data even when run on commodity hardware.

7.1 Introduction

The shift from traditional to cyber-physical grids relies on the deployment of Advanced Metering Infrastructures (AMIs) in which communication-enabled meters share data with the utility's head-end and are remotely controlled. In this context, the strict coupling between threats' cyber and physical dimensions (that can possibly result in human losses or physical damage [76]) demands for appropriate defense mechanisms. As Stuxnet[111] taught us, malicious activity designed to hide its malicious behavior can be carried out during years before being detected.

Despite the limited number of real attacks documented so far, a considerable number of possible attack vectors has been uncovered [182]. Specification-based Intrusion Detection Systems (IDSs) [39, 188], the main defense mechanism proposed so far for this domain, detect malicious activity by means of deviations from defined behavior. Such IDSs usually require a considerable amount of manual labor by a security expert in order to tune them to specific installations [39]. At the same time, they do not provide a comprehensive protection against all possible adversary goals. As an example, they might distinguish messages that comply with a given protocol from messages that do not, but might fail in distinguishing whether a message that does not violate the protocol is sent by an intact or a compromised device.

Challenges Kush et al. [165] claim traditional IDSs cannot be used effectively in these environments without major modifications and they mention nine challenges, four of which are taken into account in this paper: *scalability*, *adaptiveness*, *network topology* and *resource-constrained end devices*. As discussed in [39], AMIs consist of several independent networks whose overall traffic cannot be observed by a centralized IDS. Hence, the IDS should process data in a distributed fashion in order to embrace the different networks composing the AMI. Furthermore, the processing capacity of a centralized IDS would be rapidly exhausted by the big, fluctuating volume of data generated by AMIs' devices. To this end, the IDS should also process data in a parallel fashion in order to cope with the volumes of data and detect malicious activity timely. It should be noted that existing privacy regulations play an important role when it comes to the information accessed to spot malicious activity. As discussed in [191], fine-grain consumption readings reveal detailed information about household activities and could be used to blackmail public figures [114]. For this reason, while being interested in detecting malicious activity, the utility maintaining the AMI might not have access to underlying information owned by energy suppliers. Hence, the IDS should be able to detect malicious activity while relying on partial evidence (i.e., while accessing a limited set of traffic features). Finally, the IDS should avoid expensive per-site customization by providing an efficient way to specify how to detect malicious activities.

Contributions We present *METIS*[1], an Intrusion Detection framework that addresses these challenges by employing a two-tier architecture and the data streaming processing paradigm [235]. *METIS* has been designed giving particular attention to the detection of malicious activity carried out by adversaries that want to go unnoticed. The challenge in the detection of such malicious activity lies in that suspicious traffic proper of a given adversary goal can be caused by both legitimate and malicious factors. We provide the following contributions:

1. A two-tier architecture that provides a scalable traffic analysis that can be effective while (possibly) relying on a limited set of traffic features. Its two-tier architecture eases the system expert interaction (who can model the traffic features affected by an adversary goal by means of *Bayesian Networks*) and allows for complementary detection mechanisms such as specification-based and signature-based ones to be integrated in the framework.

2. A prototype implementation programmed using Storm [236], a state of the art Stream Processing Enginge used mainstream applications (such as twitter).

3. One of the first evaluations based on data extracted from a real-world AMI and focusing on *energy exfiltration* attacks in which the adversary aims at

[1] Named after the mythology figure standing for good counsel, advice, planning, cunning, craftiness, and wisdom.

stealing energy consumption information from AMI users. The evaluation studies both the detection capabilities of the framework and its applicability while relying on commodity hardware. To the best of our knowledge, detection of such attacks has not been addressed before.

The paper is structured as follows. We introduce some preliminary concepts in Section 7.2. In Section 7.3 we overview the *METIS*' architecture while we discuss its implementation in Section 7.4. An example showing how the framework is applied to the energy exfiltration use-case is presented in Section 7.5. We present our evaluation in Section 7.6, survey related work in Section 7.7 and conclude in Section 7.8.

7.2 Preliminaries

7.2.1 Advanced Metering Infrastructure model

We consider a common AMI model, composed of two types of devices: *Smart Meters* (SMs), in charge of measuring energy consumption and exchanging event messages such power outage alarms or firmware updates, and *Meter Concentrator Units* (MCUs), in charge of collecting such information and forwarding it to the utility head-end. Different network topologies exist in real-world AMIs (e.g., point-to-point, hierarchical or mesh ones). In order to encompass all possible networks and represent AMIs that can evolve over time, we consider a generic network, in which SMs are not statically assigned to specific MCUs.

Among the messages that are exchanged by the AMI's devices, two are of particular interest with respect to the use-case that will be introduced in the following: Energy Consumption Request (ECReq) messages, sent by MCUs, and Energy Consumption Response (ECResp) messages, sent by SMs. Such messages are used to retrieve energy consumption and can be exchanged several times per day.

7.2.2 Intrusion Detection in Advanced Metering Infrastructures

AMIs are characterized by their slow evolution and limited heterogeneity. That is, they are composed by a limited set of device types and their evolution is dictated by small (and often planned) steps (e.g., deployment of a new meter, replacement of a broken meter, and so on). Given a time frame that ranges from days to months, such evolution is "slow" and thus enables for detection techniques, such as anomaly-based ones, building on machine learning mechanisms. Nevertheless, the same evolving nature demands for a continuous learning that evolves together with the AMI (thus addressing the *adaptiveness* and *network topology* challenges discussed in [165]).

As introduced in Section 7.1, distributed and parallel network traffic analysis should be employed in order to embrace the different networks that compose AMIs while coping with the large and fluctuating volume of data produced by their devices. The distinct deployment options for an IDS in this domain can be character-

ized in a spectrum. At one extreme, the analysis could be performed by the utility head-end system. In this case, the devices should be instructed to report their communication exchanges to the head-end (at least, the ones that are required to detect a given attack). On the other extreme, the computation could be performed by the AMI's devices themselves, as investigated recently in [209]. This option would also be limited by the computational resources of the devices. Intermediate solutions could rely on a dedicated sensing infrastructure that runs the analysis together with the utility head-end system, as discussed in [127]. To our advantage, relying on the data streaming processing paradigm simplifies the deployment of an AMI defense framework to the requirement of providing a set of nodes (sensing devices or servers) that embraces the possible existing networks of the AMI. We refer the reader to [129] for a detailed discussion about how data streaming applications can be deployed at arbitrary number of nodes (thus addressing the *scalability* challenge discussed in [165]).

7.2.3 Adversary model

Several types of attacks can be launched against AMIs. On one hand, attacks such as Denial of Service (DoS) or Distributed Denial of Service (DDoS) are meant to be noticed (i.e., they impose a challenge in their mitigation rather than detection). On the other hand, more subtle attacks can be carried out by adversaries that want to go unnoticed. This second type of attacks (imposing a challenge in their detection) are the main target of *METIS*. Such adversaries could be interested in installing a malicious firmware that, while leaving the device's communication unaffected, would allow them to use the AMI as a communication medium [127]. At the same time, a malicious firmware could also be installed to lower bills by reducing the consumption readings reported by the meters (causing an *energy theft* attack [181]).

Energy Exfiltration use-case In this scenario, the adversary aims at stealing energy consumption information from AMI users. As discussed in [191], fine-grained consumption readings collected over a sufficiently large period reveal detailed information about household activities and could be used to blackmail public figures [114]. Given our AMI model, such malicious activity can be carried out after successfully logging into an MCU or by deploying a (malicious) MCU replica and collecting energy consumption readings over a certain number of days. The subtle nature of this attack lies in that suspicious exchanges of ECReq and ECResp messages can be caused not only by the adversary, but also by legitimate factors (e.g., noisy communication between devices, unreachable devices, and so on).

7.2.4 Data Streaming

A stream is defined as an unbounded sequence of tuples $t_0, t_1, \ldots$ sharing the same schema composed by attributes $\langle A_1, \ldots, A_n \rangle$. Data streaming *continuous queries* are defined as graphs of operators. Nodes represent operators that consume and produce tuples, while edges specify how tuples flow among operators.

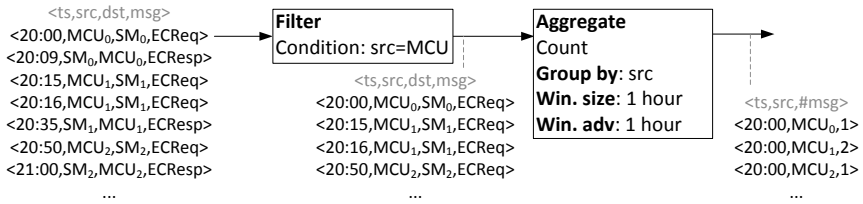


Fig. 7.1: Sample query that computes the number of messages forwarded by each MCU during the last hour. The figure includes the abstract schema and a set of sample tuples for each stream.

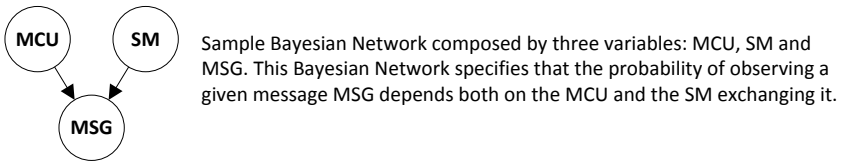


Fig. 7.2: Sample Bayesian Network.

Operators are distinguished into *stateless* (e.g., *Filter*, *Map*) or *stateful* (e.g., *Aggregate*, *EquiJoin*, *Join*), depending on whether they keep any evolving state while processing tuples. Due to the unbounded nature of streams, stateful operations are computed over *sliding windows* (simply windows in the remainder), defined by parameters *size* and *advance*. In this context, we focus on time-based windows. As an example, a window with size and advance equal to 20 and 5 time units, respectively, will cover periods $[0, 20)$, $[5, 25)$, $[10, 30)$ and so on.

The generic schema of the streams generated by the AMI's devices is composed by attributes $\langle ts, src, dst, msg \rangle$, specifying the timestamp ts at which message msg is forwarded by source src to destination dst. In the remainder, we use the terms tuple and message interchangeably when referring to the devices' communication. Figure 7.1 presents a sample query that computes the number of messages forwarded by each MCU during the last hour for a given set of input tuples (also shown in the figure).

7.2.5 Bayesian Networks

Bayesian Networks (BNs) provide a probabilistic graphical model in which a set of random variables (and their dependencies) are represented by means of a Directed Acyclic Graph. Given two random variables A and B, a directed edge from A to B specifies that the latter is conditioned by the former [117]. The conditional probability $P(B = b_j | A = a_i)$ represents the probability of observing b_j given that a_i has already been observed. Figure 7.2 presents a sample Bayesian Network in relation with our AMI model.

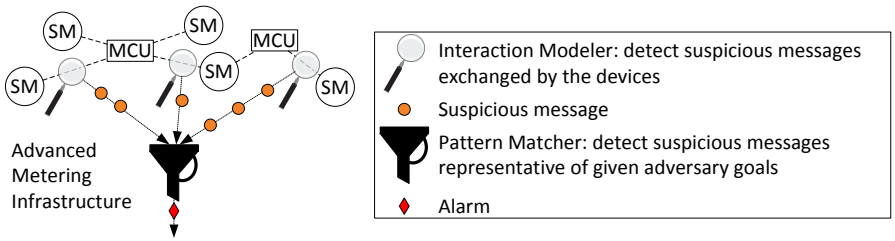


Fig. 7.3: Overview of *METIS* two-tier architecture.

7.3 *METIS* - Overview

This section overviews *METIS*' architecture and presents how adversary goals can be specified by the system expert. Multiple adversary goals can be specified at the same time. For the ease of the exposition, we provide examples that focus on our energy exfiltration use-case.

7.3.1 Architecture overview

Millions of messages are generated on a daily basis by the AMI's devices. Such messages carry heterogeneous information related to energy consumption, energy quality and power outages, among others. If we put ourselves in the role of the system expert, it might be hard to specify how evidence of a given adversary goal could be detected while processing such traffic as a whole. The work required by the system expert can be simplified by splitting it into two narrower tasks: (i) specify how an adversary goal could affect the interaction of certain types of devices (possibly belonging to different networks) and (ii) specify the pattern of suspicious interactions that could be observed over a certain period of time. This decomposition would also ease the deployment of a scalable distributed and parallel traffic analysis. The interaction of the devices could be studied close to the devices themselves (i.e., embracing the different networks of an AMI and monitoring the potentially huge amounts of traffic in parallel). Based on these observations, we designed *METIS* to analyze the AMI traffic by means of two tiers: the *Interaction Modeler* and the *Pattern Matcher* (as presented in Figure 7.3). Among its benefits, this two-tier architecture allows for other IDS to be *plugged* into the framework (e.g., by replacing the provided *Interaction Modeler* with a specification-based IDS such as [39]).

Interaction Modeler This tier analyzes the messages received and sent by each device and relies on anomaly-based detection to distinguish the ones that are expected from the *suspicious* ones.

The anomaly-based technique employed by the *Interaction Modeler* distinguishes between expected and *suspicious* messages based on the probability of observing them. It should be noticed that such probability evolves over time and is potentially influenced by several factors. As an example, the probability of observing an ECReq message could depend on the MCU forwarding it, on the SM

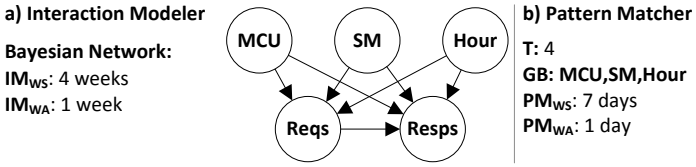


Fig. 7.4: Input provided by the system expert for *METIS*' *Interaction Modeler* and *Pattern Matcher*

receiving it, on the quality of the communication between these two devices, and so on.

If we tackle this aspect from the system expert point of view, it is desirable to have an intuitive way of specifying with traffic features should be taken into account for a given adversary goal. To our advantage, Bayesian Networks (BNs) provide an effective and graphical way of representing such features and their inter-dependencies. At the same time, BNs can also be automatically translated into data streaming queries, as we discuss in Section 7.4.2.

Since *METIS* relies on the data streaming processing paradigm, probabilities are maintained over a window of size IM_{WS} and advance IM_{WA} (specified by the system expert), thus coping with the evolving nature of AMIs. IM_{WS} represents the period of time during which traffic should be observed in order to have representative probabilities. IM_{WA} specifies the amount of information that should be discarded each time the window slides. As an example, if parameters IM_{WS} and IM_{WA} are set to 12 months and 1 months, respectively, probabilities based on the traffic observed during the last year would be produced every month.

Pattern Matcher The anomaly-based detection mechanism employed by the *Interaction Modeler*, based on the probability with which messages are expected, can result in legitimate messages being considered as *suspicious*. As an example, this could happen when lossy communication between a pair of devices leads to a low expectation associated to a certain legitimate message. For this reason, the *Pattern Matcher* consumes the *suspicious* messages forwarded by the *Interaction Modeler* in order to distinguish the ones that are isolated from the ones representative of a given adversary goal, raising an *alarm* in the second case.

The system expert is required to specify how *suspicious* messages should be processed by means of four parameters. An *alarm* is raised if a threshold T of *suspicious* messages sharing the same values for the set of attributes GB are observed given a window of size PM_{WS} and advance PM_{WA}.

7.3.2 Energy exfiltration use-case

Interaction Modeler Given our adversary model for the energy exfiltration use-case, the malicious traffic would result in an unusual exchange of ECReq and ECResp messages between a pair of MCUs and SMs. Hence, the system expert could define a BN composed by two variables: Reqs (the number of ECReq mes-

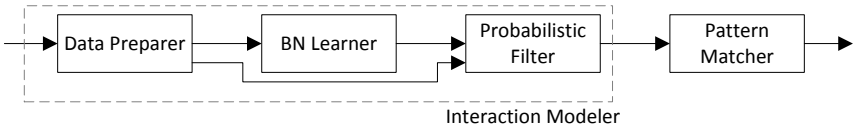


Fig. 7.5: Overview of the query created by *METIS*.

sages observed in the window) and Resps (the number of ECResp messages observed in the window), with Reqs being a conditional variable for Resps. In our model, SMs are not statically connected to MCUs. Moreover, energy consumption readings can be retrieved multiple times at different hours during each day (the hour actually depends on the MCU). For this reason, more variables could be added to the BN, as shown in Figure 7.4.a. Since SMs do not change the MCU to which they connect on a daily basis, a window of four weeks (IM_{WS}=4 weeks) updated every week (IM_{WA}=1 week) could be long enough to detect unexpected exchanges of ECReq and ECResp messages.

Pattern Matcher As discussed in Section 7.2.3, the adversary is willing to collect energy consumption readings over a certain number of days in order to infer detailed information about the victim's household activities. In this example (Figure 7.4.b), the system expert specifies that an *alarm* should be raised if at least four suspicious messages (T=4) are observed for the same MCU, SM and hour (GB=MCU,SM,Hour) given a window of size seven days (PM_{WS}=7 days) and advance one day (PM_{WA}=1 day).

7.4 Detecting anomalies by means of continuous queries

As discussed in Section 7.3.1, one of the motivations of *METIS* is to ease the system expert's interaction with the framework. For this reason, *METIS* decouples the semantics of the analysis from its actual implementation and deployment. That is, it requires the expert to specify how to detect a given adversary goal by means of a BN and a set of parameters, while it is responsible for compiling such information into a data streaming query. In the following sections we overview the processing carried out by the query, also discussing how the BN is learnt by means of data streaming operators.

7.4.1 Continuous query - overview

Both the traffic analysis of the *Interaction Modeler* and the *Pattern Matcher* are carried out by a single data streaming query compiled by *METIS*. For the ease of the exposition, we present this query by means of four modules: the *Data Preparer*, the *BN Learner*, the *Probabilistic Filter* and the *Pattern Matcher* (as presented in Figure 7.5). The first three modules perform the analysis of the *Interaction Modeler* while the last module is responsible for the analysis of the *Pattern Matcher*.

The *Data Preparer* pre-processes the information required to learn the given BN. It relies on a Filter operator to discard messages that are not relevant for the BN and on an Aggregate operator to aggregate the information based on the BN's variables. The tuples forwarded by the *Data Preparer* are consumed by the *BN Learner*, in charge of maintaining the probabilities over the window of size IM_{WS} and advance IM_{WA}. The exact number of operators that compose the *BN Learner* depends on the number of variables specified by the BN, as we discuss in the following section. The tuples produced by the *BN Learner* associate the messages observed during the given window to a certain probability. This information is processed, together with the information produced by the *Data Preparer*, by the *Probabilistic Filter*. As discussed in Section 7.2.2, the evolving nature of AMIs demands for continuous learning. For this reason, the *Probabilistic Filter* compares each tuple produced by the *Data Preparer* with its associated probability learned over the latest completed window. As an example, if parameters IM_{WS} and IM_{WA} are set to 10 and 5 time units, respectively, the window will cover periods $P_1 = [0, 10)$, $P_2 = [5, 15)$, $P_3 = [10, 20)$, and so on. Messages observed in period $[10, 15)$ would be matched with the probabilities learned during period P_1, messages observed in period $[15, 20)$ would be matched with the probabilities learned during period P_2, and so on. A tuple produced by the *Data Preparer* is forwarded by the *Probabilistic Filter* based on a probabilistic trial. As an example, if the probability learned for a certain message is 0.9, such a message will be forwarded with a probability equal to 0.1. Tuples forwarded by the *Probabilistic Filter* represent the tuples considered as *suspicious* by the *Interaction Modeler*. As discussed in Section 7.3.1, an *alarm* is raised if at least T *suspicious* messages sharing the same values for the set of attributes GB are observed given a window of size PM_{WS} and advance PM_{WA}. The *Pattern Matcher* relies on an Aggregate operator to count how many *suspicious* messages sharing the same values for the set of attributes GB are received given a window of size PM_{WS} and advance PM_{WA}. A Filter operator is used to filter only the tuples produced by the Aggregate operator whose counter is greater than or equal to T. We provide an example of the continuous query associated to the energy exfiltration use-case in Section 7.5.

7.4.2 Learning BNs by means of data streaming operators

The number of operators composing the *BN Learner* depends on the variables defined for the BN. As we discuss in the following, the ability to automatically convert a BN to a query boils down to the ability of computing the probabilities of its variables by means of data streaming operators.

Given two discrete variables X such that $supp(X) \in \{x_0, x_1, \ldots, x_m\}$ and Y such that $supp(Y) \in \{y_0, y_1, \ldots, y_n\}$ and a sequence S of observations $o_1, o_2, \ldots$ such that $o_s = \langle x_i, y_j \rangle$ and all observations belong to the same window, the conditional probability can be computed as

$$P(Y = y_j | X = x_i) = \frac{|\{o_s \in S | o_s = \langle x_i, y_j \rangle\}|}{|\{o_s \in S | o_s = \langle x_i, . \rangle\}|}$$

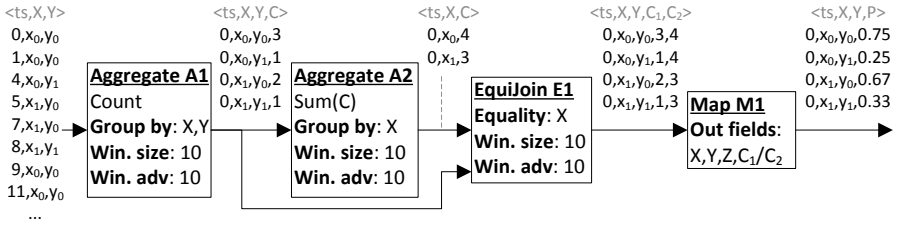


Fig. 7.6: Continuous query used to compute $P(Y|X)$. The figure includes the abstract schema and a set of sample tuples for each stream.

In order to compute such a value, we need to count the number of occurrences of each pair $\langle x_i, y_j \rangle$ and each value x_i. In terms of data streaming operators, these numbers can be maintained by two Aggregate operators. The first Aggregate operator would count the occurrences of each pair $\langle x_i, y_j \rangle$. Similarly, the second Aggregate operator would count the occurrences of each value x_i. Subsequently, values referring to the same x_i value could be matched by an EquiJoin operator and the resulting division computed by a Map operator.

Figure 7.6 presents a sample execution of the operators for a given sequence of tuples. In the example, variable X assumes values $\{x_0, x_1\}$ while variable Y assume values $\{y_0, y_1\}$. In the example, the windows' size and advance parameters are both set to 10 time units.

7.5 Energy exfiltration use-case - Sample Execution

In this section, we provide a sample execution of the continuous query compiled by *METIS*, given the BN and the parameters presented in Section 7.3.2. The query is presented in Figure 7.7. For the ease of the exposition, we focus on the messages exchanged between a single pair of MCUs and SMs, $\langle mcu_0, sm_0 \rangle$.

The *Data Preparer* module relies on its Filter operator to forward only ECReq and ECResp messages. These messages are then consumed by the Aggregate operator, in charge of counting how many ECReq and ECResp messages are exchanged between each MCU and SM and for each hour. In the example, malicious messages (injected by the adversary) are marked in red. As shown in the figure, an exchange of a single ECReq and a single ECResp message is observed twice while an exchange of two ECReq and two ECResp messages is observed only once during the month of September. Similarly, exchanges of one ECReq and one ECResp messages, two ECReq and two ECResp messages, and three ECReq and two ECResp messages are observed once during the month of October. The last two tuples produced by the Aggregate operator are marked in red since they are influenced by the malicious input messages.

The probability of observing each combination is computed by the *BN Learner* module. The probability of observing an exchange of one ECReq and one ECResp messages is 67% while the probability of observing an exchange of two ECReq

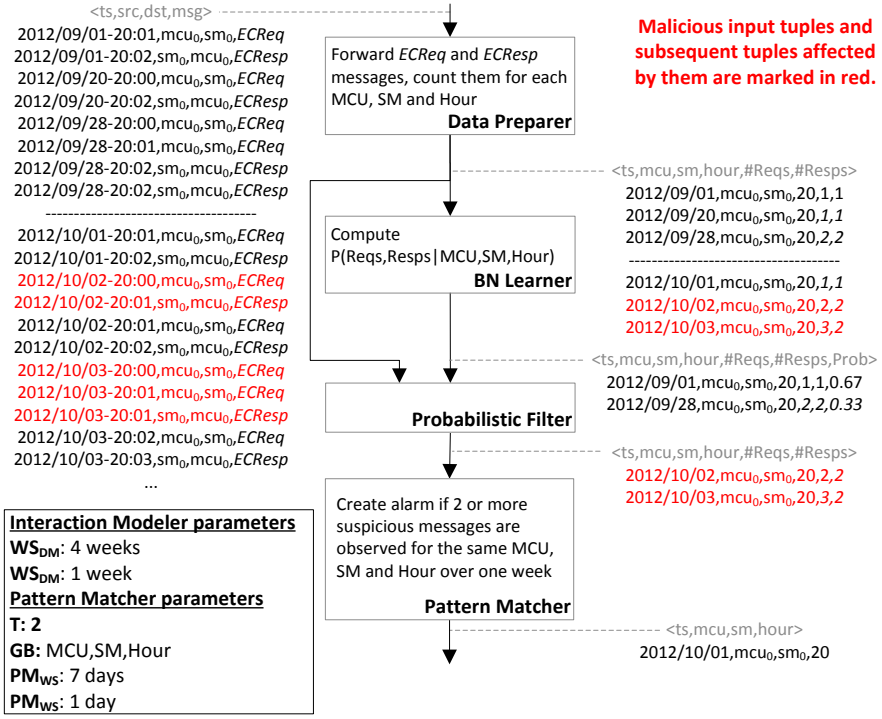


Fig. 7.7: Sample execution of the query compiled for the energy exfiltration use-case. The figure includes the abstract schema and a set of sample tuples for each stream.

and two ECResp messages is 33%. The probabilities computed by the *BN Learner* and the tuples produced by the *Data Preparer* are matched by the *Probabilistic Filter*. As discussed in Section 7.4.1, each tuple produced by the *Data Preparer* is matched with its associated probability observed in the latest completed window. In the example, tuples produced during the month of October will be matched with the probabilities observed for the month of September. Tuples $\langle 2012/09/01, mcu_0, sm_0, 20, 1, 1\rangle$, $\langle 2012/09/02, mcu_0, sm_0, 20, 2, 2\rangle$ and $\langle 2012/09/03, mcu_0, sm_0, 20, 3, 2\rangle$ have a probability of 0.33, 0.67 and 1, respectively, of being considered as suspicious. In the example, tuples $\langle 2012/09/02, mcu_0, sm_0, 20, 2, 2\rangle$ and $\langle 2012/09/02, mcu_0, Sm_0, 20, 3, 2\rangle$ are considered as suspicious and forwarded. Since the threshold T is set to two, an alarm is raised by the *Pattern Matcher*.

7.6 Energy Exfiltration use-case - Evaluation

In this section we evaluate *METIS* with respect to our energy exfiltration use-case and show that (i) it is able to detect malicious activity and that (ii) it can be leveraged by relying on commodity hardware. We first present the evaluation setup, discussing the real world AMI from which data is extracted and the attack injection methodology for the energy exfiltration attacks. We continue by presenting the detection accuracy for a given configuration of the *Interaction Modeler* and the *Pattern Matcher*, also discussing how different configurations affect their detection capabilities. Subsequently, we evaluate the processing capacity of *METIS* (in terms of throughput and latency) when executed by a server that could be deployed at the utility head-end.

7.6.1 Testbed and dataset description

METIS has been implemented on top of Storm, version 0.9.1. The continuous query (*topology* in Storm's terminology) is composed by fourteen operators. The real-world AMI used in our evaluation is composed by 300,000 SMs that communicate with 7,600 MCUs via IEEE 802.15.4 and ZigBee. The network covers a metropolitan area of 450 km^2 with roughly 600,000 inhabitants. The utility extracted data for a subset of 100 MCUs that communicate with approximately 6,500 SMs and made it available for us. The input data covers a period of six months ranging from September 2012 to February 2013. To the best of our knowledge, this dataset is free from energy exfiltration attacks. SMs are not statically linked to MCUs. At the same time, SMs appear and disappear (e.g., because of new installations or decommissioning). MCUs are in charge of collecting energy consumption readings at different hours, usually two or three times per day (the hours at which the collection happens is specific for each MCU). Due to the wireless communication, it is common for MCUs and SMs to lose messages that are thus forwarded multiple times. Each MCU has a maximum of three attempts per hour to retrieve the energy consumption of a given SM. The information kept by the utility does

not contain the exact number of messages exchanged for a given MCU, SM and day. Nevertheless, we are able to compute the probabilities with which a message is lost (and hence sent again) based on the logs stored by the MCUs. The ECReq and ECResp messages for each MCU, SM and day are simulated based on such probabilities.

In order to inject adversary traffic, we randomly pick a MCU-SM pair and, during a period that goes from seven to ten days, we inject ECReq and ECResp messages. In total, we inject 50 energy exfiltration attacks, resulting in 995 malicious messages. Note that these messages are subject to the same probability of being lost as any legitimate message. Furthermore, in order to simulate the behavior of a subtle adversary, malicious messages are exchanged at the same hour at which the MCU is actually retrieving energy consumption readings (as it would be trivial to detect an energy exfiltration attack if messages are exchanged when the MCU is not supposed to communicate).

7.6.2 Detection Accuracy

In this experiment, the BN is the one presented in Section 7.3.2. The *Interaction Modeler*'s parameters IM_{WS} and IM_{WA} are set to four weeks and one week, respectively. The *Pattern Matcher*'s window parameters PM_{WS} and PM_{WA} are set to seven days and one day, respectively. The *Pattern Matcher* is instructed to raise an alarm if at least a threshold T of five suspicious messages sharing the same values for the set of attributes MCU,SM and Hour is observed. A summary of the results is presented in Table 7.1.

AMI data	Number of attacks	50
	Number of malicious messages	995
	Overall number of messages	$4,146,327$
	Messages per day (average)	$23,743$
	Suspicious messages per day (average)	450
Interaction Modeler	Malicious messages considered as suspicious	857
	Malicious messages not considered as suspicious	138
Pattern Matcher	Number of alarms	488
	Alarms, True Positive	245
	Alarms, False Positive	243
	Detected Attacks	45

Table 7.1: Summary of the *Interaction Modeler*'s and the *Pattern Matcher*'s detection results.

During the six months covered by the data, more than 4.2 million messages are exchanged between the 100 MCUs and the $6,500$ SMs taken into account (more than $23,000$ messages on average on a daily basis). Nevertheless, a small number of approximately 450 messages are considered suspicious on average by the

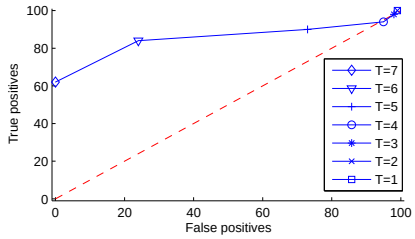


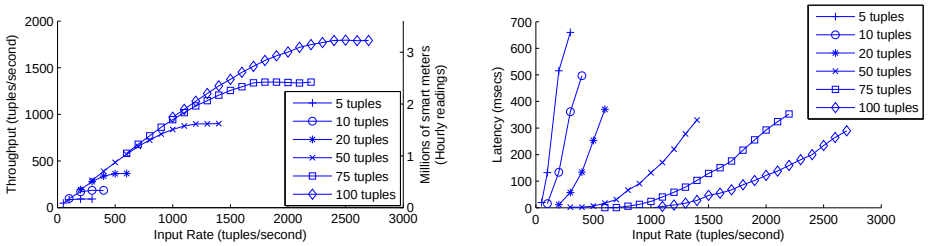
Fig. 7.8: True Positive and False Positive rates for varying thresholds T.

Interaction Modeler on a daily basis. 857 out of the 995 malicious messages are considered as suspicious. In total, 488 alarms are raised by the *Pattern Matcher*, 245 of which are related to real attacks (45 attacks are actually detected).

We say an alarm raised by the *Pattern Matcher* is a true positive (resp., false positive), if the period of time covered by its window of size PM_{WS} and advance PM_{WA} actually includes days in which malicious activity has been injected for the given MCU, SM and Hour. It should be noticed that since the window slides every day (PM_{WA} is set to one day), multiple alarms can be raised during consecutive days for one or more suspicious messages referring to a given MCU, SM and Hour. The number of false positives (243) raised during the six months period results in one or two false positives per day, on average. This number of false positives is reasonable for the system expert to use the framework (a reasonable threshold is set to no more than ten false positives per day in [174]). We further analyzed the cause of these alarms and interestingly, most of these false positive alarms are due to new smart meters that appear in the traffic. As this evaluation is based on a real deployment, we can draw the conclusion that the number of devices in this environment is not stable (meaning any assumption of the former would cause false alarms).

7.6.3 Parameters sensitivity

For a given configuration of the *Pattern Matcher*'s parameters PM_{WS}, PM_{WA} and GB, the number of attacks detected by the former depends on the threshold T (i.e., it depends on the number of days during which suspicious messages should be observed in order to raise an alarm). In this section, we present how the true positive and false positive detection rates are affected by varying the values of the threshold T. Since the *Pattern Matcher*'s Aggregate window size (PM_{WS}) is set to seven days, the experiments are run for $T = 1, \ldots, 7$. As presented in Figure 7.8, the minimum true positive rate is achieved when parameter T is equal to seven. In this case, no false positive alarms are raised by the *Pattern Matcher*. It can be noticed that the true positive rate increases to more than 80% when $T \leq 6$, while it grows to more than 90% when $T \leq 4$.



(a) Throughput (tuples/second). (b) Latency (milliseconds).

Fig. 7.9: Throughput and latency for increasing input rates and batch sizes.

7.6.4 Processing capacity

As shown above, *METIS* is able to detect the majority of the energy exfiltration attacks we injected. In this section, we show it can also cope with the large volume of events produced in a typical AMI. For that reason, we evaluate the processing capacity of *METIS* when running on a server that could be deployed at the utility's head-end, an Intel-based workstation with two sockets of 8-core Xeon E5-2650 processors and 64 GB DDR3 memory.

Among the different parameters that could influence the processing capacity of the query, the *batch* size plays a fundamental role in this context. While processing messages, a trade-off exists between the rate at which such messages can be processed and the latency imposed by the processing itself. In high-throughput systems, it is common to group tuples together in batches (of thousands or tens of thousands of tuples) in order to achieve higher throughput. Nevertheless, this is not an option in our scenario. Each pair of devices exchanges a small number of messages per hour (in the order of tens). If the analysis relies on big batches (e.g., thousands of messages), devices might not be able to log incoming and outgoing messages for the resulting large periods of time and possible attacks would thus not be detected.

Figure 7.9a presents the processing throughput for different batch sizes, from 5 to 100 tuples. As expected, increasing the batch size results in higher processing throughput. For a batch size of 100 tuples, the server is able to process approximately $2,000$ messages per second. Based on our data, each pair of MCU and SM exchanges one ECReq and one ECResp message each time energy consumption is retrieved. If the $2,000$ messages processed every second refer to the exchange of $1,000$ pairs of MCUs and SMs, the processing capacity of our prototype would enable the monitoring of more than three millions pairs of MCU and SM every hour. Figure 7.9b presents the corresponding latency (in milliseconds) for the different batch sizes. While a common pattern is observed for all batch sizes (the latency starts increasing when the throughput gets closer to its maximum), it can be noticed that the highest measured latency is of approximately 0.7 seconds. This means that the latency in the detection of an attack would depend on the frequency

with which energy consumption readings are retrieved rather than the (negligible) processing time introduced by *METIS*' analysis.

7.7 Related Work

Despite their recent deployment, a considerable number of potential attacks against AMIs has already been discussed in literature where some have even been seen in the wild [163]. The attacks range from energy theft [181], stealing of users' information [40], up to physical damage of the infrastructure [76].

As outlined in [165], traditional IDSs cannot be used effectively in these environments without major modifications. However, even though there exists a large literature on intrusion detection in general, very few systems have been developed specifically for AMIs. Several papers motivate the need for security in smart grids (where [107] is such an example); others go one step further and discuss detection mechanisms but often concentrating on other parts of the smart grid (such as attack detection in SCADA networks [68], or for process control [131]). Berthier et al. [40] discuss requirements with an outline of a possible intrusion detection architecture suitable for AMIs. To the best of our knowledge, specification-based IDSs are the main defense mechanism proposed so far for AMIs [39, 188].

One advantage with our approach is that several detection mechanisms can be used as a sensor in the first tier (the *Interaction Modeler*), meaning that the previously suggested specification-based approaches for AMIs could also be integrated into our framework. However, in this paper we instead suggested Bayesian inference in the first tier. Using Bayesian networks to model attacks merges the best properties of the signature-based approach with the learning characteristics of anomaly detection [246]. A specification-based IDS would require manual labor to tune the system to a specific installation, where a Bayesian attack model would be (relatively) easy to create for the system expert with the added benefit that we automatically can parallelize it in METIS by relying on the data streaming paradigm. Specification-based systems work best in very stable environments; in AMIs it is expected that the traffic will be more dynamic and less deterministic in the future with demand-side networks, as described in [165].

Using several tiers of sensors and analysis engines to improve the detection has been used in traditional IDSs such as [242, 210]. Our motivation for having different tiers is that they allow for the implementation of the *Interaction Modeler* to be isolated from the overall event processing. As mentioned above, this approach makes the design and implementation of the attack models easier. The second tier manages the scalability of the approach to allow for the analysis of the underlying traffic in real time.

As discussed in [40, 127], the coexistence of distinct networks within the same AMI demands for distributed traffic analysis, either by relying on the devices themselves (as recently investigated in [209]) or by relying on dedicated sensing infrastructures. To this end, the data streaming processing paradigm [235] is an optimal

candidate for AMIs traffic analysis, as explored in [223, 267, 110]. The latter is the closer to our approach, but their evaluation is not based on data from realistic AMIs.[2]

7.8 Conclusions

This work proposed *METIS*, a two-tier defense framework that eases the modeling of possible adversary goals and allows for a scalable traffic analysis by employing the data streaming processing paradigm. The proposed architecture allows for modular functionality and configurable deployment, allowing also for complementary intrusion detection systems to be integrated in the framework (e.g., by replacing the first tier). In the paper, besides describing and analyzing its design and implementation, we showed how it is possible for a system expert to model the detection of energy exfiltration attacks, a challenging adversarial goal. Moreover, through the evaluation of the use-case based on big volumes of data extracted from a real world AMI, we showed that *METIS*' analysis can achieve high detection rates, with low false alarm numbers, even when relying on commodity hardware.

It is worth pointing out that the possibility for distributed deployment of *METIS* enables for the detection of a variety of scenarios, including those whose detection is only possible through distributed evidence. The latter opens a path for new research in detecting and mitigating adversarial actions in AMIs, where for scalability and privacy purposes it can be imperative to detect unwanted situations close to the data sources, without the need to store the original data.

[2] They use the KDD Cup 99 dataset, with known problems (http://www.kdnuggets.com/news/2007/n18/4i.html) as well as lacking realistic AMI attacks.

8

Analysis of the Impact of Data Granularity on Privacy for the Smart Grid

Preamble: Relation to the Research Roadmap

One of the major threats identified in the Red Book is the loss of anonymity in online activities. Actions are recorded and stored, creating very large datasets. By analysing such datasets one can deduct very private information about individuals. This is a concern for all cyber activities, but the trend in smart environments to have sensors recording more data close to the individual emphasizes the threat. Such datasets include the health monitors that are becoming popular, the high granularity of the measurements of energy consumption readings, etc.

In this chapter, we investigate datasets of energy consumptions and possible risks of de-anonymization. In particular, the main question studied is whether data can be collected in such a way as to keep an adequate privacy level and still be useful for billing and grid operational purposes.

Tudor Valentin, Magnus Almgren, and Marina Papatriantafilou. "Analysis of the impact of data granularity on privacy for the smart grid." Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society. ACM, 2013.

Abstract

The upgrade of the electricity network to the "smart grid" has been intensified in the last years. The new automated devices being deployed gather large quantities of data that offer promises of a more resilient grid but also raise privacy concerns among customers and energy distributors.

In this paper, we focus on the energy consumption traces that *smart meters* generate and especially on the risk of being able to identify individual customers given a large anonymized dataset of these traces. This is a question raised in the related literature and an important privacy research topic. We present an overview of the current research regarding privacy in the Advanced Metering Infrastructure. We make a formalization of the problem of de-anonymization by matching low-frequency and high-frequency smart metering datasets and we also build a threat model related to this problem. Finally, we investigate the characteristics of these datasets in order to make them more resilient to the de-anonymization process.

Our methodology can be used by electricity companies to better understand the properties of their smart metering datasets and the conditions under which such datasets can be released to third parties.

8.1 Introduction

In any new domain where significantly more data starts being produced, the privacy of the customer who produces these data may be at risk. This is also the case in the new *smart grid* which is the name used for the modern electrical grid. One of the main differences between the traditional electrical grid and the new smart grid is the large number of computing and communication devices being installed in different parts of the grid and that are connected through an overlay communication network; their main purpose is to make the grid monitoring and operational processes more accurate and more efficient.

These computing and communication devices are deployed in all of the three main sections of the electrical network: the generation section, the transmission section and the distribution section. Specifically, in the distribution section, the traditional electro-mechanical meters that used to monitor the electrical energy consumed by the end customers are replaced by the new so-called *smart meters*. The smart meters, together with other devices that monitor, gather and send their data to the energy distributor's central location form the *Advanced Metering Infrastructure* (*AMI*). The AMI offers two-way communication between the central control system and the smart meters, resulting in better remote functionality of the smart meters, such as remote shut-off commands and control of demand-side electricity load and generation. Figure 8.1 presents an overview of the AMI, together with an exemplification of the different types of communication media (radio, wired, fiber-optics) and protocols used (Ethernet, Power Line Communication, ZigBee, GPRS) in suggested deployments.

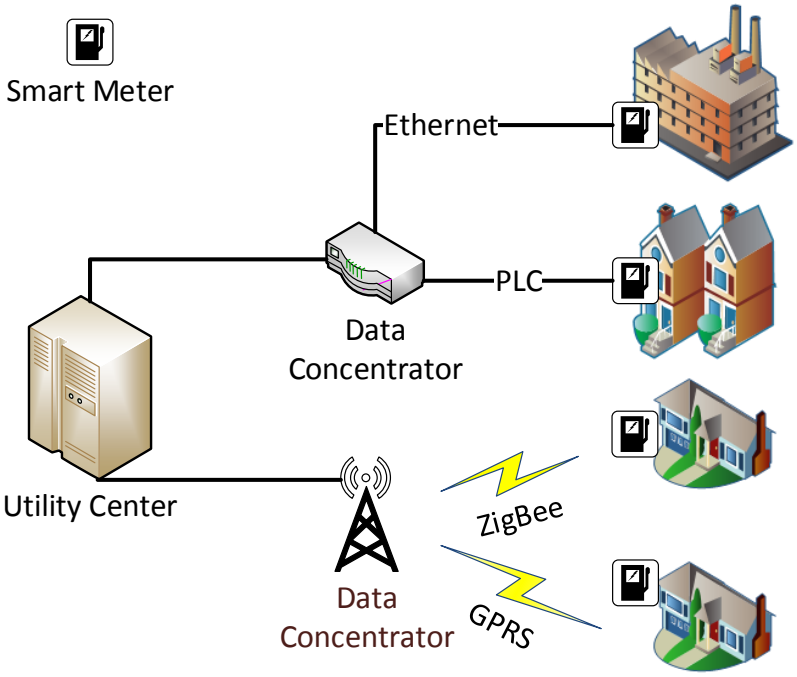


Fig. 8.1: The Advanced Metering Infrastructure (AMI)

As a consequence of the upgrade to the smart grid, significantly more data is collected and analyzed, for example in the AMI where more parameters than just the electrical energy consumed by customers are recorded, at a higher frequency than before. It is estimated that the size of the smart grid will be larger than the size of the Internet[1] and the quantity of data produced will be considerable. These data are expected to play a key role in the development of the smart grid and will improve the balance between energy production and energy consumption by making a significant contribution in improving electrical grid stability and energy efficiency.

However, there are concerns that these benefits may come at the cost of privacy: the large quantity of data produced and the granularity with which individual items are collected raise privacy concerns regarding the information that can be inferred about the lifestyle of the customers. In some countries, the debate regarding customer's privacy has even slowed down the deployment of smart meters [141].

Therefore, the main question is whether data can be collected in such a way as to keep an adequate privacy level and still be useful for billing and grid operational purposes. Using the terminology from [207], any solution should offer *anonymity* (the state of being not identifiable in a set of subjects) and also temporary *unlinkability* (the relation of two items based on the adversary's observations) of the

[1] http://news.cnet.com/8301-11128_3-10241102-54.html

customer with the quantity of electrical energy used in that specific unit of time. However, in the smart grid full unlinkability is almost impossible to be attained because the customer needs to be billed at some point for the resources used. The same goes for the *unobservability* (usage of a resource without someone to be able to observe that the resource is being used); the aggregated consumption is known at all times as the energy used for a group of customers is monitored at the substation level.

In this paper we first provide an *overview* of the current research regarding privacy in the AMI where we present some of the current privacy problems and privacy enhancing technologies proposed in the literature, motivating also the contributions presented subsequently in the paper. We construct a *formalization* of the de-anonynimization problem present in the AMI. The problem is caused by matching two types of datasets collected in the AMI, the low-frequency dataset (mainly used for billing of customers) and the high-frequency dataset (mainly used for grid operation). We build a theoretical model that describes this problem and also a threat model presenting a possible de-anonymization scenario performed by an adversary. We perform an *investigation* of the characteristics of these datasets in order to make them more resilient to the de-anonymization process. In our investigation, we concentrate on the data collected in the distribution network from the AMI where we focus mainly on the data granularity and timespan.

The rest of the paper is structured as follows: in Section 8.2 we present the general considerations of data privacy in the AMI as well as the different data types that can be collected. We give an overview of the current literature, present the main questions regarding the privacy concerns raised by AMI data and describe the characteristics of the two types of datasets mentioned earlier. Section 8.3 formally describes the de-anonymization problem, followed by the development of the theoretical framework and the threat model. Section 8.4 describes the investigation conducted and a discussion of the results obtained. This paper concludes with Section 8.5 which summarizes our results and their implications.

8.2 Data Privacy in the Advanced Metering Infrastructure

As mentioned in the previous section, the main improvement introduced by the smart grid in the distribution section is the replacement of the traditional electromechanical meters with the new smart electrical meters which are the main producers of data from the AMI. Before the smart meters, energy consumption readings were usually made every month or even less frequently, usually by a human operator visiting each customer individually, so the quantity of data gathered was not even comparable with the one today.

8.2.1 Data from the Advanced Metering Infrastructure

Data from the Advanced Metering Infrastructure are primarily used for *billing purposes* and consist of the *index of energy consumption* in kWh. The modern smart meters offer the possibility to extract much more information about the well-being of the electrical distribution network. For billing of residential customers, only the quantity of the so-called *active energy* consumed is required. For high industrial consumers the quantity of *reactive energy* used may also be billed; grid operation may require information about instantaneous values of voltage, current, active/reactive power, power outage logs, errors in the metering equipment, and much more. Table 8.1 shows a short list of useful data types that can be gathered from the AMI.

Billing data	Operational Data
Active energy	Power (active, reactive, power factor)
Reactive energy	Voltage (value, phase angle)
	Current (value, phase angle)
	Power outage logs, Alarms

Table 8.1: Data from the AMI

Efthymiou et al. [104] use the term *high-frequency data* for data used for grid operational purposes and *low-frequency data* for data used for billing purposes. We will keep the same definitions throughout this paper. Low-frequency data need to be collected seldomly (every month or every few months) but the law dictates that such data need to be identifiable to a specific customer for correct billing and to prevent fraud, both from the customer side and from the utility provider side. High-frequency data, used for grid operations, need to be collected very often (every few minutes) in order to give an accurate overview of the electrical distribution network. Previous research [43, 191] shows that fine-grained data can infer information about the lifestyle of the inhabitants such as electrical device usage patterns and presence or absence from the premises. Although the utility of these data for grid operation is evident, the privacy concerns that may be raised cannot be ignored. In an ideal case, these data should not be identifiable with a specific customer [104], but with a group of customers served by the same electrical transformer or distribution station.

8.2.2 Data usage in the Advanced Metering Infrastructure

As mentioned in the previous section, there are several different types of data that can be collected in large quantities from the Advanced Metering Infrastructure. The main consumer of these data is the Distributor System Operator (DSO), followed by other third parties, each of them having different purposes. Data can be used by the DSO for billing, processed for fraud detection, operational purposes

(grid stability and security) or marketing. Third parties (researchers, other companies, malicious entities) may also be interested in these data for benign activities (research, marketing) or for malign ones (fraud, invasion of privacy or even attacks against the critical infrastructures).

The privacy preserving techniques (PPTs) are usually implemented at a large scale by the DSO, or a legal trusted third party and at a small scale by the customers. When thinking about a specific privacy preserving technique it is important to remember the complexity of the parties that may have access and use the data produced. For example, we should be able to answer the following (not exhaustive) list of questions:

- does the PPT offer privacy protection against DSOs?
- does it offer privacy protection against third parties?
- does it provide availability of billing data?

If interested in providing privacy for its customers, the DSO may prefer to employ a solution that offers privacy protection against third parties but which first provides availability of the billing data.

The customer may prefer a solution that offers privacy protection against both the DSO and other third parties, while availability of billing data might come in a later position in the customer's priority list. Thus, the DSO's and the customer's visions of privacy might be different and even conflicting. In an ideal case, the customer's data privacy should be protected against both the DSO and other third parties.

8.2.3 Overview of smart grid privacy mechanisms in the literature

As a concept, Warren and Brandeis [251] give in 1890 the definition of "privacy" as the "right to be let alone". More recently, Pfitzmann and Hansen [207] define the terminology to be used when talking about privacy by data minimization.[2] From a legal point of view, to the best of our knowledge, there is no specific European Directive which covers smart metering data privacy. Thus, only the general European Directive, EU Data Protection Directive 95/46/EC [99], would cover these types of data. However, the German Federal Office for Information Security developed the Protection Profile for the Gateway of a Smart Metering System; closely related to this, Stegelmann and Kesdogan [234] propose an architecture called GridPriv that includes a non-trusted k-anonymity service for pseudonymised meter data.

[2]The terminology includes: anonymity, unlinkability, linkability, undetectability, unobservability, pseudonymity, identifiability, identity, partial identity, digital identity and identity management.

Siddiqui et al. [222] make an overview of some of the proposed solutions towards preserving privacy in the smart grid and divide these into the following categories: *anonymous credentials, third party escrow mechanisms, load signature moderation, smart energy gateway and privacy-preserving authentication.*

Anonymous credentials are based on blind signatures (similar to the ones used in the e-cash payment systems) and have the advantage to offer privacy protection against both DSO and third parties. The disadvantage of this solution is that it does not provide availability of billing data and it can only be used for pre-paid energy.

Third party escrow mechanisms [43, 104, 248] require the presence of a trusted third party entity whose role is to anonymize the data collected from the customers and then present it to the DSO or to aggregate the data and present it in an anonymized form. As mentioned in Section 8.2.1, Efthymiou and Kalogridis [104] present a solution based on separation of data into attributable low-frequency data, collected seldom and mainly used for billing, and anonymized high-frequency data, collected very often and used for grid operation. Each of these will be reported using a different pseudonym (one public and one private) and only the trusted third party is supposed to know the connection between the anonymous pseudonym and the public one. Their solution offers privacy protection against other third parties and also provides availability for billing data. The open question that remains is if the DSO can later recreate low-frequency data from the high-frequency and match it with the already available low-frequency data and so breaking the privacy. We will return to this question in Section 8.3.

Load signature moderation [152, 153] is a good privacy preserving method that can be used by customers. It requires the presence of an energy storage facility at the customer premises, such as an old battery from an electrical vehicle. The customer can then even out her external load signature by drawing energy from the battery in the high-load periods or by charging it during the low consumption periods or when energy is cheaper. This method offers protection both against DSOs and other third parties and also provides availability of billing data, because the Smart Meter will register only the energy used from the electricity network. However, the method has the disadvantage of requiring extra hardware.

The last two categories proposed by Siddiqui et al. [222] are *smart energy gateway* and *privacy-preserving authentication*. In the same way as load signature moderation, these also require the presence at the customer premises of a dedicated system. In the first case the system is responsible to manage data released from the smart meter on some internal rules based on the data requester, while in the second case its role is to create trusted pseudo-identities that are used in requesting different energy amounts. In the first case privacy protection and availability of billing data can be enforced by setting up proper rules; the second one can only be used in a pre-paid energy scenario.

Hiding in the crowd is another method used to preserve privacy. Borges et al. [48] present a solution based on anonymity networks in which a customer uses two different identities to send his billing data and grid-operational data. While the billing data is directly attributable to him, the grid-operational data is forwarded

to the DSO through an anonymity network, so the customer cannot be directly identified in a group of customers from the same network.

Data aggregation can also be used as a privacy-preserving solution. Before data is aggregated, one initial step in order to prevent unlawful disclosure of information is to perform mutual authentication [260, 261] between the entities involved in the process. Following this, privacy against the DSO and third parties can be obtained by using homomorphic cryptography [48, 164, 179, 261], or by adding random noise from a known distribution of zero mean [164, 179], but unfortunately aggregating methods do not provide availability of billing data and techniques based on homomorphic cryptography can be expensive on devices with reduced processing power and low resources such as the currently deployed smart meters.

Privacy enhancing techniques should also be resistant to attacks. Jawurek et al. [148] present the problem of breaking smart meter privacy by using de-pseudonymization. They propose a framework based on machine learning with support vector machines for the analysis of consumption traces and tracking consumption traces across different pseudonyms by using two linking procedures. Linking by Behaviour Anomaly (LA) tries to link a real ID to a consumption trace or two consumption traces together by correlating anomalies that happen in the same time, for example consumption spikes or blackouts. Linking by Behaviour Pattern (LB) tries to link different pseudonyms for one consumer and their method can be applied even if the consumption profiles do not overlap in time. In this paper we show that even simpler functions may also work quite well in identifying customers.

Buchmann et al. [51] show that identification of individual houses based on their energy-consumption records is possible even by using simple statistical tools such as means and standard deviations on a reduced number of data features. They show that 68% of the records coming from a set of 180 houses can be re-identified by using these simple methods.

So far we presented an overview of the current literature regarding privacy in the smart grid context. Next we will present the research papers that are close connected to our work.

Related work especially relevant to this paper:

Out of the presented papers above, the ones that are most closely related to ours are [51, 57, 104, 148]. Efthymiou and Kalogridis [104] set up the terminology on which we build our framework. Their solution is based on a trusted third party that takes care of the private IDs used in the process of high-frequency data anonymization and also of the connections between the high-frequency ID and the low-frequency one. Jawurek et al. [148] present a de-pseudonymization framework based on machine learning and are focusing mainly on anomalies in data consumption that happen in the same time. For their solution, fine-grained data is required, because such anomalies can be missed if aggregated daily or monthly values are used. Compared with their solution, we are focusing mainly on aggregated consumption where we try to identify uniqueness. Buchmann et al. [51] use simple

statistical tools on a reduced number of consumption features and also on external information sources such as physical observation of people habits. Focusing on demand-response schemes, Cárdenas et al. [57] present the problem of appropriate sampling intervals in AMI as a trade-off between keeping a good level of customer privacy and gains in the demand-response scheme properties. They focus on the economics behind this problem as a parameter into the proper sampling scheme.

8.2.4 Advanced Metering Infrastructure data characteristics and problem formulation

Summarising regarding the AMI data characteristics on the two types of active energy consumption data reported by the smart meters in AMI, high-frequency (HF) data and low-frequency (LF) data, the question that arises is how these data should be reported and gathered in order to keep an adequate level of privacy against both the DSO and third parties? The level of privacy is measured as a reduced number of uniquely identifiable customers based on these two types of reported data.

There are a number of questions to which the research community tries to find the answers:

- Can customers be identified based on their energy consumption reported by the smart meters?
- How similar are customers with each other based on their energy consumption trace?

There are three characteristics of these data that were identified in the literature that determine the privacy level: number of *pseudonyms* for the same customer used in reporting/storing data, the *timespan* of data stored by the utility provider and the *granularity* of reported/stored data. The investigation presented in Section 8.4 focuses on the last two of these characteristics and on their role in making the datasets more resilient to the de-anonymization process.

Reporting high-frequency data under different pseudonyms and making sure that connections between pseudonyms are extremely hard to find and/or known by only a trusted third party [48, 104, 164] have been proposed earlier in the research literature. Using one pseudonym can be useful, if the connection between this pseudonym and the real customer ID is secret, but reporting or storing data from the same smart meter under different pseudonyms for shorter timespans can be very efficient [48]. Although useful, generating multiple pseudonyms can be expensive for the smart meter device, because they need to create them through the use of a cryptographic algorithm, or they need to be provided when shipped from the factory.

The timespan of data stored is also very important, because longer periods of stored data for a smart meter (under the same pseudonym) can infer much more information about the energy consumption that took place. The question here is

what the window for stored data that is useful for billing/grid operation is but which is also, at the same time, privacy preserving?

The last characteristic taken into consideration is the granularity of reported and stored data. Low-frequency data must be reported in fine-grained detail for accurate billing and to prevent fraud. Customers want, naturally, only to be billed for what they consumed, and the utility company wants to know exactly how much is consumed in order to level production and to better operate the grid. Unfortunately, loss occurs in the distribution grid due to transformers and old equipment, and are taken into consideration [98]. The question is whether the reported high-frequency data can be altered in a minor way such that the modification will not affect the grid operation, but making it hard to identify each customer uniquely by, for example, making the data from different customers more uniform? Figure 8.2 presents these three characteristics in relation to the adequate privacy level that is desired.

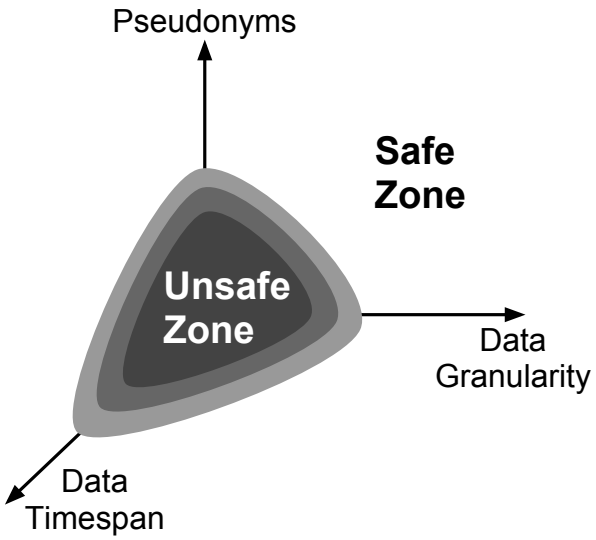


Fig. 8.2: Characteristics of AMI data

8.3 Methodology

We will now formally describe the de-anonymization process by linking a *low-frequency dataset* with a *high-frequency dataset*. We will also present a threat scenario featuring an adversary which attempts to uniquely identify as many customers as possible and learn as much as possible, for example about their habits and living conditions, using the information from the high-frequency dataset.

Based on this scenario, in Section 8.4 we will conduct an investigation using a large real dataset on which we will study the influence of data timespan and granularity in the de-anonymization process. Our methodology can be used to bet-

ter understand the limits of what is safe and what is not with regard to releasing datasets to third parties.

8.3.1 Formal framework

Assume that there exists a dataset, $\mathcal{C} = \{(identifier, timestamp, value)\}$, collected from the smart meters in an advanced metering infrastructure. This dataset contains identifiers ($identifier$) that can be used to identify individual customers, as well as high-frequency data ($value$) of the form described in Section 8.2.1, each marked with a specific $timestamp$. As mentioned, the high-frequency data can be used to infer habits of households.

There are two functions, $f_H(\cdot)$ and $f_L(\cdot)$ such that we can derive two new datasets by letting

$$\begin{cases} \mathcal{H} &= f_H(\mathcal{C}) \\ \mathcal{L} &= f_L(\mathcal{C}) \end{cases}$$

where $\mathcal{H}$ and $\mathcal{L}$ are related but have slightly different properties. In a scenario within the smart grid, $\mathcal{H}$ would be a dataset with, for example, the originally collected high-frequency data but where all customer identification would be replaced with untraceable labels (one simple way to obtain untraceable labels is to use a random label generator and check for possible collisions). This dataset could then be used for grid operation and optimization as it would not be possible to use it to identify individual customers. The set $\mathcal{L}$, on the other hand, would retain the original identifiers making it possible to identify customers but instead the data in this dataset would be aggregated (under the original identifier) so as to be less privacy invasive. This dataset could then be used for billing of monthly consumption, for example. The complete dataset, $\mathcal{C}$ is then discarded.[3]

We further assume that finding f_H^{-1} and f_L^{-1} is intractable, as information is deliberately discarded in each transform. Thus, if an adversary obtained either $\mathcal{H}$ or $\mathcal{L}$ it would be difficult to recreate $\mathcal{C}$ and each dataset in isolation would not be interesting. This is similar in vein to the indirect assumptions for the solution presented by Efthymiou and Kalogridis [104].

However, as $\mathcal{H}$ and $\mathcal{L}$ originate from the same dataset, we assume that there exists another function, $g(\cdot)$, such that $\mathcal{H}'\mathcal{L}' = g(\mathcal{H})$.[4] The data in $\mathcal{H}'\mathcal{L}'$ would retain the identifying labels from $\mathcal{H}$ and be aggregated in a similar fashion to $\mathcal{L}$. If we could then link any entries between these two datasets, $\mathcal{H}'\mathcal{L}' \sim \mathcal{L}$, we might partially be able to recreate $\mathcal{C}$ by relabeling the entries in $\mathcal{H}$. The problem, though, is that many of the aggregated values in $\mathcal{L}$ may not be unique but would be the same across a number of customers meaning that we cannot easily infer which labels should be linked as there will be a set of possible matches. Intuitively, we would expect customers with a very uncommon behavior to maybe be re-identified

[3] The complete dataset, $\mathcal{C}$, might not ever exist if the transforms are run continuously in the smart meters.

[4] The existance of $g(\cdot)$ would depend on how $f_H(\cdot)$ and $f_L(\cdot)$ were constructed. Based on our survey of existing methods, we say that it is likely $g(\cdot)$ exists.

but that a majority of customers would belong to clusters that behave in a similar nature and thus not be uniquely identified.

Formally, the question we would like to answer is whether it is at all possible to link these two datasets, given a large realistic scenario. If so, we want to measure how well $\mathcal{C}$ can be recreated and provide boundaries on what an adversary can achieve if she has access to both $\mathcal{H}$ and $\mathcal{L}$. Can we limit the information gained by the adversary by changing the properties of either of these datasets, for example by using more pseudonyms or storing less data as discussed in Section 8.2.4?

8.3.2 Adversarial strategy

A possible adversarial strategy algorithm, that is also implied by the previous literature, is presented in Algorithm 1, and it is used to derive the associated adversary model. The adversary gets hold of two sets of data, one containing high-frequency ($\mathcal{H}$) data and one containing low-frequency data ($\mathcal{L}$) with the properties described in Section 8.3.1. The individual smart meters that produced these data are labeled differently in these two datasets; to simplify the presentation, we assume each smart meter has only one identifier in each of the sets, being equivalent to using only one pseudonym. The algorithm can easily be extended with sets of more pseudonyms.

As stated above, we assume the adversary wants to be able to recreate the dataset $\mathcal{C}$, where she can label the high-frequency data with the identity of the individual customers from the low-frequency dataset. By analyzing the low-frequency datasets, she tries to find as many "unusual" customers as possible, i.e. customers that at some point in time have data values that differ from the norm so that she can create a link between $\mathcal{H}'\mathcal{L}'$ and $\mathcal{L}$. In the algorithm, this is performed in the function `findLink()`. This analysis can range in its sophistication. In the first version of the algorithm, we have chosen to implement a method that only looks for unique values in a time period to show that even a relatively simple and fast analysis can be surprisingly efficient. As we will show in Section 8.3.3, the simplicity of the function also allows us to model it as a game of *balls and bins* so that we can estimate the probabilities of the success of the scenario.

Note that our discussion so far has been of a general nature; the datasets can contain a diverse set of data as described in Section 8.2.1. However, in the following we are going to concentrate on consumption traces. These types of datasets are often used in the literature (please see Table 8.2 for an overview).

8.3.3 Probabilistic framework and analysis

In this section, we model the adversarial strategy algorithm in a probabilistic framework to be able to reason formally about the adversary's capabilities and possibilities of success in de-anonymizing customers. Given the properties of the `function findLink()` shown in Algorithm 1 it is possible to model the algorithm as a *game of balls and bins* [190]. In the following discussion, we assume the datasets contain energy consumption data e.g. kWh consumption indexes.

Algorithm 1: Adversarial strategy algorithm

```
Requirement: adversary has obtained H and L
Goal: recreate as much as possible of C
begin algorithm
    create H'L' = g(H);
    while IDlink = findLink(H'L', L) do
        recreate one entry in C;
        remove identified trace from H'L' and L
    end
end
begin function findLink
    /* Version 1: find *unique* consumption traces
       in a time period in L                          */
    foreach timeperiod j in L do
        if any unique consumption traces exists then
            extract identifying ID from L;
            find corresponding entry in H'L';
            extract identifying ID' from H'L';
            return <ID, ID'>;
        end
    end
    /* no more links can be made                      */
    return false;
end
```

The energy index data from m_j smart meters (balls) in one time period, $j \in T$, can be sorted into a set of n different intervals (bins), where the width of the bins corresponds to a range of energy consumption units (multiples of kWh). We let the width of the bins be an integer, w, that can vary from 1 to W. The number of bins is then $n = \frac{M}{w}$, where $M = max(m_T)$ is the maximum index consumption value for all the time periods considered. At each round, the number of balls in all the bins is equal with the number of balls at the beginning of the round e.g. $\sum_{i=1}^{n} m_{w_i} = m_j$.

Any ball that falls alone in a bin is considered to be uniquely identified and it is removed. This is then repeated; each round of the game uses data from a time period where index data for the m_j smart meters exist. The game ends when either all the balls are removed or when there exists no more time periods with new data. The percent of eliminated balls at the end of the game is then equivalent to the percent of uniquely identifiable consumption indexes from that specific dataset.

In the analysis of the bins and balls game presented by Mitzenmacher and Upfal [190] the probability that a bin receives a number of r balls when m balls are thrown independently and uniformly at random into n bins is given as a Poisson distribution of mean $\frac{m}{n}$.

$$P[a\,bin\,has\,r\,balls] = \frac{e^{-\frac{m}{n}} \times (\frac{m}{n})^r}{r!} \tag{8.1}$$

Note that we assume a Poisson distribution of the balls into bins. This will be further discussed in Section 8.4.2. In our specific case with $r = 1$ (only one ball per bin so that we can identify the customer), the probability above becomes the following.

$$P[a\,bin\,has\,1\,ball] = e^{-\frac{m \times w}{M}} \times \frac{m \times w}{M} \tag{8.2}$$

For the Poisson case, the number of balls in each bin must be independent random variables. In our case, the number of balls in the last bin is known as we have m balls and we know the number of balls in the first $n-1$ bins. Corollary 5.9 from [190] states the following.

Corollary 8.3.1 *[190] Any event that takes place with probability p in the Poisson case takes place with probability at most $pe\sqrt{m}$ in the exact case.*

Thus, any event that happens with a small probability in the Poisson case also happens with a small probability in the exact case, where balls are thrown into bins [190] and justifies the Poisson analysis for the bins and balls game. The *expected number of bins with only* 1 *ball* becomes the following.

$$E[bins\,with\,1\,ball] = e^{-\frac{m \times w}{M}} \times m \tag{8.3}$$

Let the number of balls available at the beginning of the game be m_0. If we consider two consecutive rounds in the game, the expected number of balls m_j at the beginning of round j can be computed as:

$$m_j = m_{j-1} - E_{j-1}[bins\,with\,1\,ball] \tag{8.4}$$

where if we substitute the expression for the expected value:

$$m_j = m_{j-1} \times (1 - \exp{(-\frac{m_{j-1} \times w}{M})}) \tag{8.5}$$

The game ends when either all balls have been removed from the game ($m_j = 0$) or all the time periods with available data, T, have been used ($j > T$).

The adversary would win the game when the percentage of extracted balls is above a specific threshold, λ, meaning that a large percentage of the smart meters have been identified uniquely ($\frac{\sum_{1 \leq j \leq T} m_j}{m_0} > \lambda$). The utility company wins the game when the percentage of uniquely identified smart meters is low (m_T is very close to m_0). By investigating the parameters (m, λ, w), we can explore the limits of the capabilities of the adversary to make sure that she cannot identify a large set of customers.

8.4 Evaluation study

As mentioned in Section 8.2.4, the *granularity* and *data timespan* play an important role in the data de-anonymization process. We investigate these two characteristics by using a *simulation* based on the probabilistic framework presented in Section 8.3.3 and an *evaluation* based on a dataset, described in Section 8.4.1. We expect to identify the influence of the characteristics in the context of the adversarial strategy algorithm presented in Section 8.3.2. The last part of this section presents a discussion of the results obtained.

8.4.1 Description of the dataset

To see how well the adversarial strategy algorithm works in a real setting we use a dataset consisting of smart meter readings from a large number of consumers in a medium-sized city. The original data have hourly smart meter index readings for a period of seven non-contiguous months. The data originates from a range of smart meters serving very small consumers (summer cottages) to large consumers (industrial customers). The data have gone through a two-step anonymization process, once by the utility provider and once by us, to make sure it is not possible to physically identify any customers in the set. Each record has the ¡ID_{anon}, $timestamp$, $value$¿ format. The *timestamp* and the index *value* remain in clear. The data from each smart meter in the set can be identified by a unique numerical identifier (ID_{anon}), that remains the same over the seven months. This is equivalent of having a single pseudonym for each of the smart meters for the whole time period.

As the data comes from a real AMI, where problems with missing values sometimes exist, we also sanitized the data by creating a smaller set where we removed a number of collection artifacts. Mainly, we removed any smart meters that had gaps in the hourly reporting (values lost), double conflicting records for the same timestamp or decreasing index values for increasing timestamp values.

Dataset	Number of meters	Number of readings
Kalogridis et al. [104]	N/A	N/A
Jawurek et al. [148]	53	281,112
Buchmann et al. [51]	180	60,480
Daisuke and Cárdenas [180]	108	$^{\star}1,890,000$
Tudor et al. (this paper)	19,334	99,355,998

Table 8.2: Datasets from AMI
The * value was estimated based on values in [180]

After the sanitization process, the dataset contained $19,334$ unique smart meters with $99,355,998$ hourly energy consumption readings. This set is considered to be the high-frequency dataset ($\mathcal{H}$). From this dataset, we then created the low-frequency dataset ($\mathcal{L}$), which is similar for each customer to the energy consumption values printed in the electrical bill. This resulted in $4,156,810$ daily values and $135,338$ monthly values. As can be seen from Table 8.2, our dataset is significantly larger than the ones previously used in literature.

8.4.2 The Poisson distribution assumption

In the game of bins and balls presented in Section 8.3.3 we assume that the balls are thrown independently and uniformly at random so that they can be modeled as a Poisson distribution.

The balls signify specific smart meters. These smart meters belong to the same households with the same number of people with habits that will probably not change on a monthly basis. Regardless of what bin a ball lands in for a round, the theoretical model assumes that it is equally likely that the ball falls into any of the bins in the next round. However, in the real case it is likely that the energy consumption pattern would be somewhat similar across months, so that it is more likely that the ball falls into a bin close to the bin from the last month. We say that the balls in the real case are somewhat *sticky* as they tend to favor, across months, bins that are closely located.

This also implies that if two balls fell into the same bin one month, it is likely that they will do so also the following month in the real case. For that reason, we expect that the identification process, using `function findLink()` in Algorithm 1, will be more difficult in the real case compared to the probabilistic model.

Furthermore, the customers may not be divided uniformly at random across the consumption bins. A typical household in Europe hosts about 2.3 people[5]. Our dataset reflects a large number of such domestic customers, agglomerated in the low consumption zone, while the probabilistic model assumes a random spread in

[5] http://appsso.eurostat.ec.europa.eu/nui/show.do?dataset=ilc_lvph01&lang=en

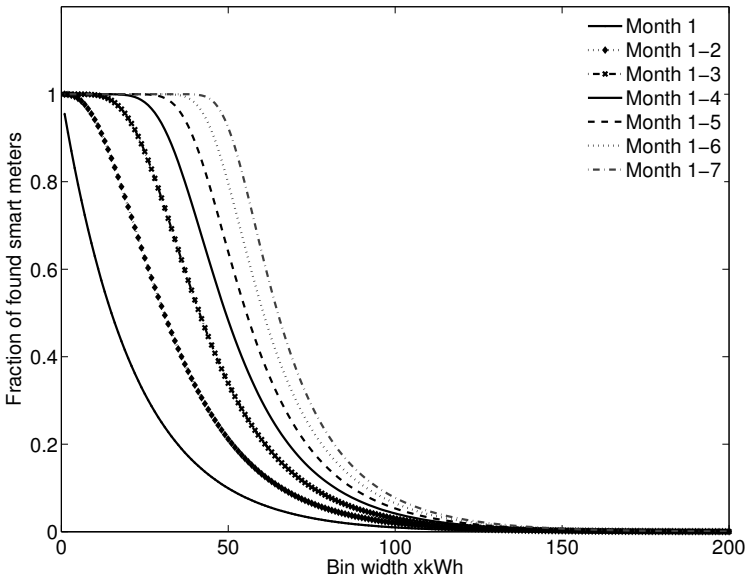


Fig. 8.3: Fraction of unique smart meters - seven months of data - estimation case

the bins. This also strengthens our expectations regarding the difficulty to identify a majority of the customers in the real case.

Even with these assumptions, the formal framework let us reason about the characteristics of the AMI datasets and their influence on the de-anonymization process.

8.4.3 Results from the probabilistic framework

The *estimation* run of the adversarial strategy algorithm is based on the formulas presented in Section 8.3.3 with the parameters adapted to match the dataset of real consumption traces. The initial number of smart meters is selected to be 19,334 in order to exactly match the number present in the dataset, and the number of rounds in the estimation is the same as the number of time periods in the dataset – seven for the monthly case and 30 for the daily case. The granularity is varied from 1 to 200 kWh. The value for M is selected to be the same as in the dataset and is the highest index value for the time periods considered. The expected number of identified smart meters (balls) at each round is computed by using Relation (8.3), while Relation (8.4) is used to compute the number of remaining smart meters (balls) at the beginning of each round.

Figure 8.3 presents the fraction of uniquely identified smart meters in the monthly case (seven time periods) obtained in the estimation, while varying the granularity with which energy consumption index is reported and the available time periods.

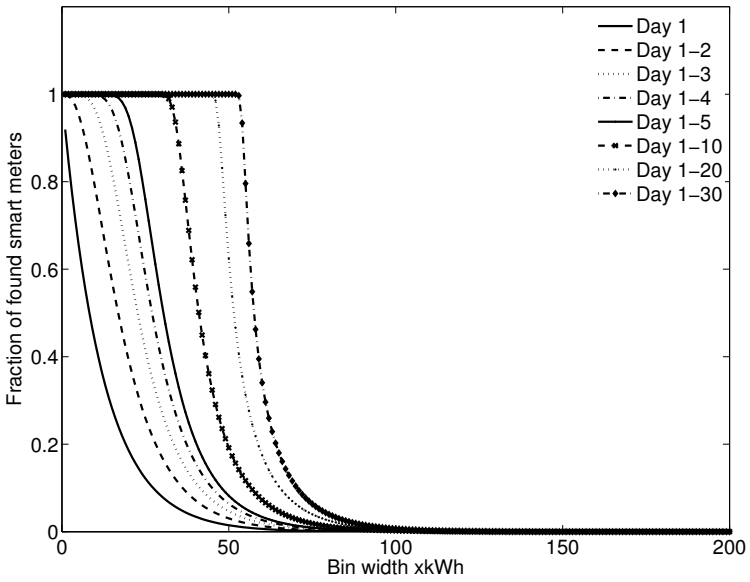


Fig. 8.4: Fraction of unique smart meters - 30 days of data - estimation case

Similarly, Figure 8.4 presents the estimation results for the daily case, using 30 time periods (only periods 1-5, 10, 20 and 30 are presented in the figure).

8.4.4 Results of the adversarial strategy algorithm

An *evaluation* of the effectiveness of the adversarial strategy algorithm is performed, for two characteristics presented in Section 8.2.4, data granularity and timespan. The starting number of smart meters is 19,334 and the number of monthly values is 135,338. There are seven time periods, equivalent to the seven months of data and the maximum index value is computed from the dataset.

Figure 8.5 presents the fraction of the uniquely identified smart meters by using the dataset presented in Section 8.4.1 for the monthly case, while varying the granularity of the reported energy consumption index from 1 to 200 kWh. The figure presents only the results from 1 to 50 kWh, after this point the values continue on what seems to be a linear trend.

The simulation is repeated by using daily datasets, the equivalent of one month of recordings (30 days - $593,830$ values), where the first month of data is used. Similarly to the monthly case, the granularity is varied between 1 and 200 kWh. Figure 8.6 shows the results based on the dataset for a granularity between 1 and 50 kWh.

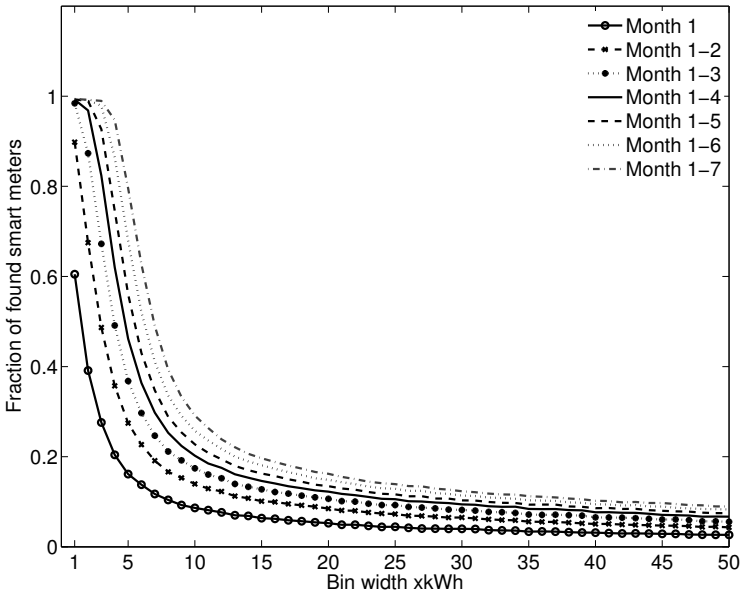


Fig. 8.5: Fraction of unique smart meters - seven months of data - dataset case

8.4.5 Dicussion of results

Table 8.3 presents the expected number of uniquely identified smart meters at each round of the simulation and the evaluation of the adversarial strategy algorithm in the monthly case for a granularity of 1 kWh. The difference between the simulation and the evaluation is that in the case of the simulation more smart meters are identified in the first round compared with the case of the evaluation based on the dataset. This makes the identification in the next rounds easier because a smaller number of smart meters needs to be divided into bins so the probability of having more than one smart meter in a bin decreases. This result can be explained through the assumptions that are discussed in Section 8.4.2. For the 1 kWh monthly case, the estimation ends after three rounds, when all the smart meters are identified. In the evaluation case, the algorithm runs for all the seven rounds, and 125 smart meters remain unidentified at the end, but the percent of identified smart meters is still above 99%.

Table 8.4 holds the results for the 10 kWh monthly case and we can observe that in the simulation case the algorithm took one more round compared to the 1 kWh case, but the percent of identified smart meters at the end is still 100%. The evaluation results for the 10 kWh monthly case show that the number of unidentified smart meters after all the rounds is 13,706 and the percent of identified smart meters is 29.1%. This is a much better result than for the 1 kWh monthly case.

Figure 8.5 shows that varying the granularity under which data are reported can drastically reduce the fraction of identified smart meters. For example, reporting

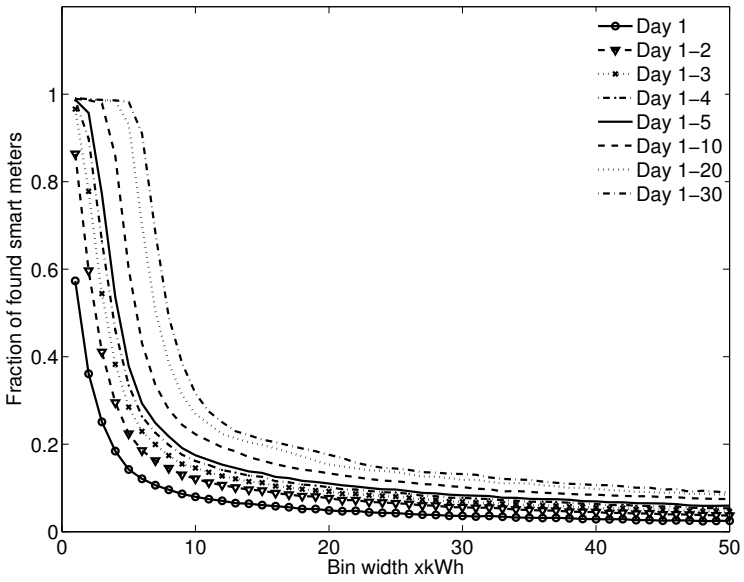


Fig. 8.6: Fraction of unique smart meters - 30 days of data - dataset case

the index without the last digit, at a 10 kWh scale, can reduce in this case the percent of identified smart meters at under 10% for one period and under 30% for all periods. This result justifies a reporting scheme in which electrical energy consumed is rounded to the next 10 kWh value, before it is reported and billed, instead of being reported with 1 kWh accuracy. This will provide a good and cheap anonymity solution for the other 70% of the customers, in the case that everyone opts for such a reporting scheme. The same result can be observed in Figure 8.6 where for the daily reporting with 10 kWh granularity, the percent of identified customers is brought down to almost 10% for one period and to almost 40% for all periods. This simple reporting solution offers a better degree of privacy, but it may still not be feasible in regions where the law requires that energy reporting should be done with kWh accuracy.

We can see that high-frequency datasets contain so much information so that the re-identification process is possible even with simple means. In our analysis we have assumed that the adversary would have access to the complete high-frequency dataset and the complete low-frequency dataset.

The evaluation results show that reporting energy consumption indexes with kWh accuracy makes the datasets prone to re-identification, because a large percent of the customers can be identified uniquely, solely based on their energy consumption. The results closely tie together the granularity and the timespan of the data and show their common effect in the re-identification process. They show that reducing the granularity used for reporting consumption data can be a very simple

	Newly found smart meters		Total found smart meters %	
Time period	**Simulation**	**Evaluation**	**Simulation**	**Evaluation**
m_1	18461	11698	95.4%	60.5%
m_2	871	5655	99.9%	89.7%
m_3	2	1669	100 %	98.3%
m_4	0	155	100 %	99.1%
m_5	0	11	100 %	99.2%
m_6	0	11	100 %	99.3%
m_7	0	10	100 %	99.3%
Total	19334	19209	100 %	99.3%

Table 8.3: Expected number of identified smart meters for a reporting granularity of 1 kWh

and beneficial solution that increases the privacy level of the datasets. Results from Tables 8.3 and 8.4 strengthen this assumption and show a significant decrease of the percent of uniquely identified smart meters from 99.3% to 29.1%, for a decrease of granularity from 1 kWh to 10 kWh. Also, Figures 8.5 and 8.6 show that further reduction of the granularity may significantly reduce the percent of identified customers, making the datasets more resilient to the de-anonymization process.

As a general consideration, data timespan and granularity should be taken into consideration before releasing any AMI consumption data to third parties, as these two characteristics greatly influence the anonymity of the datasets.

8.5 Conclusion

It is almost unquestionable that the smart grid will produce more and more data regarding the electrical energy consumed and the well-being of the electrical grid. Harnessing and processing these large quantities of data will make the electrical grid more resilient to faults, provide a better balance between the production and the consumption, but as we saw, these datasets also raise privacy concerns.

In this paper we presented an overview of research regarding smart grid data privacy. We constructed a formalization of the problem of de-anonymizing AMI data by matching two different types of smart metering datasets. We take into account two main properties of smart metering data: the granularity of the data reported and its timespan. We argue that these two, together with the number of pseudonyms used in the reporting process play a significant role in a three-way balance towards obtaining better customer anonymity. We consider a class of adversarial strategies that can be formulated as combinatorial and probabilistic problems and used it to evaluate characteristics of these datasets (granularity and timespan)

	Newly found smart meters		Total found smart meters %	
Time period	**Simulation**	**Evaluation**	**Simulation**	**Evaluation**
m_1	12182	1670	63.0%	8.6%
m_2	6029	1027	94.1%	13.9%
m_3	1093	671	99.8%	17.4%
m_4	30	543	100 %	20.2%
m_5	0	487	100 %	22.7%
m_6	0	579	100 %	25.7%
m_7	0	651	100 %	29.1%
Total	19334	5628	100 %	29.1%

Table 8.4: Expected number of identified smart meters for a reporting granularity of 10 kWh

in an investigation process towards better resilience against the de-anonymization process; our results show that this process should be taken into consideration before releasing AMI datasets. Future research directions include extending the theoretical framework and the adversarial strategy model and also to be able to limit the theoretical maximum number of customers that can be identified. Related research issues refer to billing models; it is interesting to investigate the possibilities and limitations in managing trade-offs between customer incentives for improving their usage of electricity and privacy issues regarding the data in the billing system, as these two imply different needs in the granularity of the data.

Conclusion

The objective of this deliverable is to dive deep into the research questions that are pursued in the *SysSec* project in the context of smart environments. We present a selection of the results obtained in the project during the last four years. This will help established researchers but also new PhD students in system security understand the special security challenges faced in smart environments.

We have selected the research results presented in the included chapters based on their importance for smart environments, as well as their contributions to the research areas defined by the Red Book [178]. We have shown contributions to methodologies to analyze firmwares to find vulnerabilities, patching software, and detecting attacks. We have also described risks with the large datasets that are being collected.

This last deliverable thus highlights open research questions in the area of smart environments and current problems being investigated by European researchers, and *SysSec* partners in particular. In that way, the deliverable complements the others in the series. In the first deliverable, we surveyed the state-of-the-art of security in sensor networks. We went into details about cryptography, key management, authentication, localization, clock synchronization, clustering, routing and aggregation. We explained the importance of these services and gave an overview of the state-of-the art of secure algorithms for the services. We also presented a view on the role of self-stabilization in secure systems for wireless sensor networks.

In the second deliverable, we presented one example system demonstrating the complexity of domains in the smart environment, namely that of the connected car. The deliverable gave an overview of European research, standardization efforts as well as how traditional mechanisms can be used in this environment. These include internal separation of traffic, the use of message authentication codes (MAC) to guarantee traffic integrity, firewalls both for external traffic and for internal traffic implemented in gateway ECUs, use of intrusion detection systems, use of certifi-

cates for identification of various devices (vehicles, road-side objects, drivers and ECUs) and the problems with distributing revocation lists (CRLs).

With the third deliverable, we presented a new complex smart environment, namely that of the smart grid. The objective of the deliverable was to first give a crash course for a computer scientist of security issues related to the smart grid and then survey ongoing research related to these problems. Naturally, this deliverable included sections on networking technologies and issues that they imply as part of the infrastructure. As very large datasets, also including customer data, are produced to control the smart grid, the report also included sections for privacy concerns, as well as for scalable data processing in the smart grid with a focus on security processing; besides, intrusion detection has a natural part in these contexts.

This final deliverable, together with the previous deliverables in this series, will give the reader a broad survey of ongoing research as well as concrete results and discussions of important open issues and research problems for continuation of research in a few selected areas. It is our hope that these deliverables will strengthen the future European research into system security and smart environments.

Bibliography

[1] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[2] Audit PHP Configuration Security Toolkit.

[3] Binwalk: Binary image signature checker. http://code.google.com/p/binwalk/.

[4] BYTE Magazine nbench benchmark. http://www.tux.org/~mayer/linux/bmark.html.

[5] Define of backdoor string in DLink DI-524 UP GPL source code. https://gist.github.com/ccpz/6960941.

[6] Google Custom Search Engine API.

[7] Internet Census 2012 – Port scanning /0 using insecure embedded devices. http://internetcensus2012.bitbucket.org.

[8] OsmocomBB. http://bb.osmocom.org/trac/.

[9] SHODAN – Computer Search Engine. http://www.shodanhq.com.

[10] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[11] IEEE Standard Test Access Port and Boundary-Scan Architecture, 1990. IEEE Standard. 1149.1-1990.

[12] IEEE-ISTO 5001 - 2003 the nexus 5001 forum standard for a global embedded processor debug interface. IEEE - Industry Standards and Technology Organization, December 2003.

[13] CWSandbox, 2008. http://www.cwsandbox.org.

[14] CVE-2009-2629: Buffer underflow vulnerability in nginx. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2629, 2009.

[15] Slashdot: Backdoor found in TP-Link routers, March 2013.

[16] Download statistics for the wemo android application, February 2014. http://xyo.net/android-app/wemo-JJUZgf8/.

[17] Download statistics for the wemo iOS application, February 2014. http://xyo.net/iphone-app/wemo-J1QNimE/.

[18] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[19] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 2003.

[20] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow integrity. In *Proc. of the 12th ACM conference on Computer and communications security*, CCS'05, 2005.

[21] A. D. Aczel and J. Sounderpandian. *Complete Business Statistics*. McGraw-Hill, sixth edition, 2006.

[22] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. of the IEEE Symposium on Security and Privacy*, S&P'08.

[23] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. of the 18th Usenix Security Symposium*, USENIX-SS'09, 2009.

[24] H. Allcott. Rethinking real-time electricity pricing. *Resource and Energy Economics*, 2011.

[25] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the international conference on Management of data*, 2013.

[26] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*, pages 283–300, Feb. 2011.

[27] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proc. of the 27th Intern. Conf. on Software Engineering (ICSE)*, 2005.

[28] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, 2011.

[29] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[31] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: toward a stateful network protocol fuzZEr. In *Proceedings of the 9th international conference on Information Security*, ISC'06, 2006.

[32] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th International Symposium on Software testing and analysis*, ISSTA'10, 2010.

[33] Z. Basnight, J. Butts, J. L. Jr., and T. Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76 – 84, 2013.

[34] L. Bass, N. Brown, G. M. Cahill, W. Casey, S. Chaki, C. Cohen, D. de Niz, D. French, A. Gurfinkel, R. Kazman, et al. Results of CMU SEI Line-Funded Exploratory New Starts Projects. 2012.

[35] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Symposium on Network and Distributed System Security*, NDSS '09. The Internet Society, 2009.

[36] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A View on Current Malware Behaviors. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[37] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX 2005 Annual Technical Conference*, ATEC '05.

[38] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[39] R. Berthier and W. H. Sanders. Specification-based intrusion detection for advanced metering infrastructures. In *IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2011.

[40] R. Berthier, W. H. Sanders, and H. Khurana. Intrusion detection for advanced metering infrastructures: Requirements and architectural directions. In *Smart Grid Communications (SmartGridComm), First IEEE International Conference on*, 2010.

[41] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proc. of the 12th conference on USENIX Security Symposium*, SSYM'03, 2003.

[42] A. Blanco and M. Eissler. One firmware to monitor'em all. *Ekoparty*, 2012.

[43] J.-M. Bohli, C. Sorge, and O. Ugus. A privacy model for smart metering. In *Communications Workshops (ICC), 2010 IEEE International Conference on*, pages 1 –5, may 2010.

[44] H. Bojinov, E. Bursztein, and D. Boneh. Embedded management interfaces: Emerging massive insecurity. In *Blackhat 2009 Technical Briefing / whitepaper*, 2009.

[45] H. Bojinov, E. Bursztein, and D. Boneh. Xcs: Cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 420–431, New York, NY, USA, 2009. ACM.

[46] H. Bojinov, E. Bursztein, E. Lovett, and D. Boneh. Embedded management interfaces: Emerging massive insecurity. *BlackHat USA*, 2009.

[47] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management*, 2001.

[48] F. Borges, L. Martucci, and M. Mühlhäuser. Analysis of privacy-enhancing protocols based on anonymity networks. 2012.

[49] E. Bosman, A. Slowinska, and H. Bos. Minemu: The Worlds Fastest Taint Tracker. In *Proc. of 14th International Symposium on Recent Advances in Intrusion Detection*, RAID 2011, 2011.

[50] J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. EFPL Press, 2011.

[51] E. Buchmann, K. Böhm, T. Burghardt, and S. Kessler. Re-identification of smart meter data. *Personal and Ubiquitous Computing*, 17(4):653–662, 2013.

[52] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

[53] C. Cadar, D. Dunbar, and D. Engler. KLEE unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[54] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc.s of the 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'08, 2008.

[55] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[56] M. Callau-Zori, R. Jiménez-Peris, V. Gulisano, M. Papatriantafilou, Z. Fu, and M. Patiño-Martínez. STONE: a stream-based DDoS defense framework. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[57] A. A. Cárdenas, S. Amin, and G. Schwartz. Privacy-aware sampling for residential demand response programs. 2012.

[58] Carna Botnet. Internet census 2012, port scanning /0 using insecure embedded devices, 2012. http://internetcensus2012.bitbucket.org/paper.html.

[59] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, 2002.

[60] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Impl.*, OSDI'06, 2006.

[61] L. Cavallaro, P. Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA'08, 2008.

[62] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile

Malware Analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[63] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the ACM SIGMOD international conference on Management of data*, 2003.

[64] S. Checkoway, D. McCoy, D. Anderson, B. Kantor, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analysis of Automototive Attack Surfaces. In *Proceedings of the USENIX Security Symposium*, San Francisco, CA, August 2011.

[65] K. Chen. Reversing and exploiting an Apple firmware update. *BlackHat USA*, 2009.

[66] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, 2005.

[67] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proc. of 14th USENIX Security Symposium*, SSYM'05, 2005.

[68] S. Cheung, B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes. Using model-based intrusion detection for SCADA networks. In *Proceedings of the SCADA security scientific symposium*, 2007.

[69] P. C. Ching, Y. Cheng, and M. H. Ko. An in-circuit emulator for TMS320C25. *IEEE Transactions on Education*, 37(1):51–56, 1994.

[70] V. Chipounov and G. Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), Paris France, April 2010*, Paris, France, 2010.

[71] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in vivo multi-path analysis of software systems. In *Proc. of 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[72] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012.

[73] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[74] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proc of the 2003 Conf. on Programming languages design and implementation*, POPL'03.

[75] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proc. of the 20th ACM Symposium on Operating Systems Principles*, SOSP'05, 2005.

[76] M. Costache, V. Tudor, M. Almgren, M. Papatriantafilou, and C. Saunders. Remote control of smart meters: friend or foe? In *Computer Network Defense (EC2ND), Seventh European Conference on*, 2011.

[77] A. Costin. Hacking Printers for Fun and Profit.

[78] A. Costin. PostScript(um): You've Been Hacked.

[79] A. Costin and A. Francillon. Short Paper: A Dangerous 'Pyrotechnic Composition': Fireworks, Embedded Wireless and Insecurity-by-Design. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, WiSec '14. ACM, 2014.

[80] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of the 7th USENIX Security Symposium*, SSYM'98, 1998.

[81] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.

[82] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, 1982.

[83] A. Cui. Embedded Device Firmware Vulnerability Hunting with FRAK. *DefCon 20*, 2012.

[84] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[85] A. Cui, M. Costello, and S. J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Proceedings of the 20th Symposium on Network and Distributed System Security*, NDSS '13. The Internet Society, 2013.

[86] A. Cui, Y. Song, P. V. Prabhu, and S. J. Stolfo. Brave New World: Pervasive Insecurity of Embedded Network Devices. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 378–380, Berlin, Heidelberg, 2009. Springer-Verlag.

[87] A. Cui and S. J. Stolfo. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-area Scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 97–106, New York, NY, USA, 2010. ACM.

[88] A. Cui and S. J. Stolfo. Defending embedded systems with software symbiotes. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 358–377, Berlin, Heidelberg, 2011. Springer-Verlag.

[89] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, New York, NY, USA, 2008. ACM.

[90] CWE/SANS. TOP 25 Most Dangerous Software Errors. www.sans.org/top25-software-errors, 2011.

[91] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 21st Symposium on Network and Distributed System Security*, NDSS '14. The Internet Society, 2014.

[92] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the USENIX Security Symposium*, Washington, DC, August 2013.

[93] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 463–478, Berkeley, CA, USA, 2013. USENIX Association.

[94] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[95] G. Delugré. Closer to metal: Reverse engineering the broadcom netextreme's firmware. HACK.LU 2010.

[96] G. Delugré. Closer to metal: reverse-engineering the Broadcom NetExtreme's firmware. *Hack.lu*, 2010.

[97] J. DeMott. The evolving art of fuzzng. DEFCON 14, http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.odp, 2005.

[98] J. Dickert, M. Hable, and P. Schegner. Energy loss estimation in distribution networks for planning purposes. In *PowerTech, 2009 IEEE Bucharest*, pages 1–6, 2009.

[99] E. Directive. 95/46/ec of the european parliament and of the council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal of the EC*, 23:6, 1995.

[100] L. Duflot, Y.-A. Perez, and B. Morin. What if You Can'T Trust Your Network Card? In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 378–397, Berlin, Heidelberg, 2011. Springer-Verlag.

[101] K. Dunham. A fuzzy future in malware research. *The ISSA Journal*, 11(8):17–18, 2013.

[102] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 291–304, New York, NY, USA, 2013. ACM.

[103] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its

Security Applications. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 605–620, Berkeley, CA, USA, 2013. USENIX Association.

[104] C. Efthymiou and G. Kalogridis. Smart grid privacy via anonymization of smart metering data. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 238 –243, oct. 2010.

[105] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, Mar. 2008.

[106] Elias Levy (Aleph One). Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[107] G. N. Ericsson. Cyber security and power system communicationessential parts of a smart grid infrastructure. *Power Delivery, IEEE Transactions on*, 2010.

[108] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*, OSDI '06, 2006.

[109] B. Eshete, A. Villafiorita, and K. Weldemariam. Early Detection of Security Misconfiguration Vulnerabilities in Web Applications. In *Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security*, ARES '11, pages 169–174, Washington, DC, USA, 2011. IEEE Computer Society.

[110] M. A. Faisal, Z. Aung, J. R. Williams, and A. Sanchez. Securing advanced metering infrastructure using intrusion detection system with data stream mining. In *Intelligence and Security Informatics*. Springer, 2012.

[111] N. Falliere, L. O. Murchu, and E. Chien. W32. stuxnet dossier. Technical report, Symantec Corporation, 2011.

[112] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier, 2011.

[113] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, 1997.

[114] FORWARD Consortium, White book: Emerging ICT threats. http://www.ict-forward.eu/media/publications/forward-whitebook.pdf.

[115] Freescale Semiconductor, Inc. MC1322x Simple Media Access Controller Demonstration Applications User's Guide, 9 2011. Rev. 1.3.

[116] Freescale Semiconductor, Inc. MC1322x Simple Media Access Controller (SMAC) Reference Manual, 09 2011. Rev. 1.7.

[117] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian Network Classifiers. *Machine Learning*, 1997.

[118] J. Gama and P. P. Rodrigues. Stream-based electricity load forecast. In *Knowledge Discovery in Databases: PKDD*. Springer, 2007.

[119] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE'09, 2009.

[120] M. Gegick, L. Williams, J. Osborne, and M. Vouk. Prioritizing software security fortification through code-level metrics. In *Proc. of the 4th ACM workshop on Quality of protection*, QoP'08. ACM Press, Oct. 2008.

[121] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'05, 2005.

[122] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[123] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of The ACM*, pages 40–44, 2012.

[124] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, NDSS'08, 2008.

[125] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, 2011.

[126] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward Secure Embedded Web In-

terfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[127] D. Grochocki, J. H. Huh, R. Berthier, R. Bobba, W. H. Sanders, A. A. Cárdenas, and J. G. Jetcheva. AMI threats, intrusion detection requirements and deployment recommendations. In *Smart Grid Communications (SmartGridComm), IEEE Third International Conference on*, 2012.

[128] GTM RESEARCH. White paper, The Emergence of Meter Data Management (MDM): A Smart Grid Information Strategy Report, 2010.

[129] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on*, 2012.

[130] Y. Guo, M. Pan, Y. Fang, and P. P. Khargonekar. Decentralized Coordination of Energy Utilization for Residential Households in the Smart Grid. *IEEE Transactions on Smart Grid*, 2012.

[131] D. Hadiosmanovic, D. Bolzoni, P. Hartel, and S. Etalle. MELISSA: Towards Automated Detection of Undesirable User Actions in Critical Infrastructures. In *Computer Network Defense (EC2ND), Seventh European Conference on*, 2011.

[132] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of USENIX Security'13*, Washington, DC, August 2013. USENIX.

[133] Y. Han, S. Liu, X. Su, and Z. Hu. A dynamic analysis system for Cisco IO based on virtualization. In *Multimedia Information Networking and Security (MINES), 2011 Third International Conference on*, pages 330–332, 2011.

[134] HDMoore. Security Flaws in Universal Plug and Play: Unplug, Don't Play, 2013.

[135] C. Heffner. littleblackbox – Database of private SSL/SSH keys for embedded devices.

[136] C. Heffner. Breaking SSL on Embedded Devices, December 2010.

[137] C. Heffner. Reverse Engineering a D-Link Backdoor, October 2013.

[138] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding Software License Violations Through Binary Code Clone Detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 63–72, New York, NY, USA, 2011. ACM.

[139] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[140] J. Hirsch and K. Bensinger. Toyota settles acceleration lawsuit after $3-million verdict. Los Angeles Times, October 25, 2013.

[141] R. Hoenkamp, G. B. Huitema, and A. J. de Moor-van Vugt. The neglected consumer: the case of the smart meter rollout in the netherlands. *Renewable Energy Law and Policy Review*, 2011(4):269–282, 2011.

[142] IEEE Computer Society. *IEEE 802.15.4, Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, June 2006. ISBN 0-7381-4996-9.

[143] Independent Security Evaluators. SOHO Network Equipment (Technical Report), 2013.

[144] IOActive. Critical DASDEC Digital Alert Systems (DAS) Vulnerabilities, June 2013.

[145] IOActive. stringfighter – Identify Backdoors in Firmware By Using Automatic String Analysis, May 2013.

[146] IOActive. Critical Belkin WeMo Home Automation Vulnerabilities, February 2014.

[147] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[148] M. Jawurek, M. Johns, and K. Rieck. Smart metering de-pseudonymization. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 227–236. ACM, 2011.

[149] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX 2002 Annual Technical Conference*, ATEC '02.

[150] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[151] R. Kaksonen. A functional method for assessing protocol implementation security. Technical Report 448, VTT, 2001.

[152] G. Kalogridis, R. Cepeda, S. Denic, T. Lewis, and C. Efthymiou. Elecprivacy: Evaluating the privacy protection of electricity management algorithms. *Smart Grid, IEEE Transactions on*, 2(4):750 –758, dec. 2011.

[153] G. Kalogridis, C. Efthymiou, S. Denic, T. Lewis, and R. Cepeda. Privacy for smart meters: Towards undetectable appliance load signatures. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 232 –237, oct. 2010.

[154] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

[155] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS'11, 2011.

[156] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, pages 46–53, New York, NY, USA, 2007. ACM.

[157] C.-F. Kao, I.-J. Huang, and H.-M. Chen. Hardware-software approaches to in-circuit emulation for embedded processors. *Design Test of Computers, IEEE*, 25(5):462–477, 2008.

[158] E. Kermany, H. Mazzawi, D. Baras, Y. Naveh, and H. Michaelis. Analysis of advanced meter infrastructure data of water consumption in apartment buildings. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013.

[159] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, 2003.

[160] A. Kirchner. Data Leak Detection in Smartphone Applications. Master thesis, Vienna University of Technology.

[161] J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digit. Investig.*, 3:91–97, 2006.

[162] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[163] KrebsonSecurity. FBI: Smart Meter Hacks Likely to Spread. http://krebsonsecurity.com/2012/04/fbi-smart-meter-hacks-likely-to-spread/, April 2012.

[164] K. Kursawe, G. Danezis, and M. Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *Privacy Enhancing Technologies*, pages 175–191. Springer, 2011.

[165] N. Kush, E. Foo, E. Ahmed, I. Ahmed, and A. Clark. Gap analysis of intrusion detection in smart grids. In *Proceedings of the 2nd International Cyber Resilience Conference*, 2011.

[166] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[167] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[168] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[169] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis &

transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, 2004.

[170] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for Linux. In *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS-2010.

[171] Y.-H. Lee, Y. W. Song, R. Girme, S. Zaveri, and Y. Chen. Replay debugging for multi-threaded embedded software. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 15–22, 2010.

[172] L. Li and C. Wang. Dynamic analysis and debugging of binary code for security applications. In *4th International Conference on Runtime Verification (RV) 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 403–423. Springer, 2013.

[173] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 2008.

[174] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer networks*, 2000.

[175] B. Lohrmann and O. Kao. Processing smart meter data streams in the cloud. In *Innovative Smart Grid Technologies (ISGT Europe), 2nd IEEE PES International Conference and Exhibition on*, 2011.

[176] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[177] P. D. Marinescu and C. Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *Proc. of the 2012 International Conference on Software Engineering*, ICSE'12, pages 716–726, June 2012.

[178] E. Markatos and D. Balzarotti, editors. *The Red Book: A Roadmap for Systems Security Research*. The SysSec Consortium, August 2013.

[179] F. Mármol, C. Sorge, O. Ugus, and G. Pérez. Do not snoop my habits: preserving privacy in the smart grid. *Communications Magazine, IEEE*, 50(5):166 –172, May 2012.

[180] D. Mashima and A. A. Cárdenas. Evaluating electricity theft detectors in smart grid networks. In *Research in Attacks, Intrusions, and Defenses*, pages 210–229. Springer, 2012.

[181] S. McLaughlin, D. Podkuiko, and P. McDaniel. Energy theft in the advanced metering infrastructure. In *Critical Information Infrastructures Security*. Springer, 2010.

[182] S. McLaughlin, D. Podkuiko, S. Miadzvezhanka, A. Delozier, and P. McDaniel. Multi-vendor penetration testing in the advanced metering infrastructure. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[183] C. Melear. Emulation techniques for microcontrollers. In *Wescon/97. Conference Proceedings*, pages 532–541, 1997.

[184] P. C. Messina, R. D. Williams, and G. C. Fox. *Parallel computing works !* Parallel processing scientific computing. Morgan Kaufmann, San Francisco, CA, 1994.

[185] T. Mikkola, E. Bunn, P. Hurri, G. Jacucci, M. Lehtonen, M. Fitta, and S. Biza. Near real time energy monitoring for end users: Requirements and sample applications. In *Smart Grid Communications (SmartGridComm), IEEE International Conference on*, 2011.

[186] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33:32–44, Dec 1990.

[187] C. Miller. Battery firmware hacking. *BlackHat USA*, 2011.

[188] R. Mitchell and I.-R. Chen. Behavior-Rule Based Intrusion Detection Systems for Safety Critical Smart Grid Applications. *Smart Grid, IEEE Transactions on*, 2013.

[189] Mitre. Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org/, 2011.

[190] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[191] A. Molina-Markham, P. Shenoy, K. Fu, E. Cecchet, and D. Irwin. Private memoirs of a smart meter. In *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*, pages 61–66. ACM, 2010.

[192] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86

binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, 2009.

[193] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 packets over IEEE 802.15.4 networks (RFC 4944). Technical report, IETF, September 2007. http://www.ietf.org/rfc/rfc4944.txt.

[194] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP'07. IEEE Computer Society, 2007.

[195] C. Mulliner, N. Golde, and J.-P. Seifert. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, USA, August 2011.

[196] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE'06, 2006.

[197] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proc. of PLDI'09*, 2009.

[198] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the 3rd Intern. Conf. on Virtual Execution Environ.*, VEE, 2007.

[199] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, NDSS'05, 2005.

[200] V. H. Nguyen and L. M. S. Tran. Predicting vulnerable software components with dependency graphs. In *Proc. of the 6th International Workshop on Security Measurements and Metrics*, MetriSec'10. ACM Press, Sept. 2010.

[201] K. Nohl, D. Evans, S. Starbug, and H. Plötz. Reverse-engineering a cryptographic RFID tag. In *Proceedings of the 17th conference on Security symposium*, pages 185–193, Berkeley, CA, USA, 2008. USENIX Association.

[202] Nvidia. CUDA – Compute Unified Device Architecture Programming Guide. 2007.

[203] OpenwallProject. John the Ripper password cracker. http://www.openwall.com/john/.

[204] Oracle Utilities Meter Data Management. http://www.oracle.com/us/industries/utilities/046533.pdf.

[205] OWASP. Top 10 Vulnerabilities, 2013.

[206] Y.-A. Perez and L. Duflot. Can you still trust your network card? CanSecWest 2010.

[207] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. *URL: http://dud. inf. tu-dresden. de/literatur/Anon_Terminology_v0*, 34, 2010.

[208] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '06, 2006.

[209] M. Raciti and S. Nadjm-Tehrani. Embedded cyber-physical anomaly detection in smart meters. In *Critical Information Infrastructures Security*. Springer, 2013.

[210] S. A. Razak, S. Furnell, N. Clarke, and P. Brooke. A Two-Tier Intrusion Detection System for Mobile Ad Hoc Networks A Friend Approach. In *Intelligence and Security Informatics*, Lecture Notes in Computer Science. Springer, 2006.

[211] Redwire LLC. Econotag: MC13224V development board w/ on-board debugging. http://www.redwirellc.com/store/node/1.

[212] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: testing drivers without devices. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 279–292, Berkeley, CA, USA, 2012. USENIX Association.

[213] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *CC'08/ETAPS'08: Proc. of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, 2008.

[214] V. Roussev. Data Fingerprinting with Similarity Digests. In *IFIP Int. Conf. Digital Forensics*, pages 207–226, 2010.

[215] B. Schlich. Model checking of software for microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4):36:1–36:27, Apr. 2010.

[216] P. H. Schmitt and B. Weiß. Inferring invariants by symbolic execution. In B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY'07)*, volume 259 of *CEUR Workshop Proceedings*, pages 195–210. CEUR-WS.org, 2007.

[217] F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 851–862, New York, NY, USA, 2013. ACM.

[218] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[219] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, 2005.

[220] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of USENIX Annual Technical Conference*, 2012.

[221] Y. Shin and L. Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, SESS'11, 2011.

[222] F. Siddiqui, S. Zeadally, C. Alcaraz, and S. Galvao. Smart grid privacy: Issues and solutions. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1 –5, 30 2012-aug. 2 2012.

[223] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna. Adaptive rate stream processing for smart grid applications on clouds. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, 2011.

[224] S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 23–40, Berlin, Heidelberg, 2012. Springer-Verlag.

[225] A. Slowinska and H. Bos. The Age of Data: Pinpointing Guilty Bytes in Polymorphic Buffer Overflows on Heap or Stack. In *Proc. of the 23rd Annual Computer Security Applications Conference*, ACSAC'07, 2007.

[226] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys '09: Proc. of the 4th ACM European conf. on Computer systems*, 2009.

[227] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of NDSS*, 2011.

[228] A. Slowinska, T. Stancescu, and H. Bos. Body Armor for Binaries: preventing buffer overflows without recompilation. In *Proceedings of USENIX Annual Technical Conference*, 2012.

[229] D. Song, D. Brumley, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *In Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[230] A. Sotirov. Modern exploitation and memory protection bypasses. USENIX Security invited talk, www.usenix.org/events/sec09/tech/slides/sotirov.pdf, August 2009.

[231] Spike. http://www.immunitysec.com/resources-freesoftware.shtml.

[232] S. Sridhar, J. S. Shapiro, and E. Northup. HDTrans: An open source, low-level dynamic instrumentation system. In *Proc. of the 2nd Intern. Conf. on Virtual Execution Environ.*, 2006.

[233] T. Stancescu. BodyArmor: Adding Data Protection to Binary Executables. Master's thesis, VU Amsterdam, 2011.

[234] M. Stegelmann and D. Kesdogan. Gridpriv: A smart metering architecture offering k-anonymity. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 419 –426, june 2012.

[235] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream pro-

cessing. *ACM SIGMOD Record*, 2005.
[236] Storm project. http://storm.incubator.apache.org/. Accessed: 2014-03-10.
[237] J. V. Stough. distributed-python-for-scripting – DistributedPython for Easy Parallel Scripting.
[238] StreamBase. http://www.streambase.com.
[239] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
[240] P. Team. Design and implementation of PAGEEXEC. http://pax.grsecurity.net/docs/pageexec.old.txt, November 2000.
[241] Tjaldur Software Governance Solutions. Binary Analysis Tool (BAT).
[242] E. Tombini, H. Debar, L. Mé, and M. Ducassé. A serial combination of anomaly and misuse IDSes applied to HTTP traffic. In *Computer Security Applications Conference. 20th Annual*, 2004.
[243] A. Tridgell. rsync – utility that provides fast incremental file transfer.
[244] A. Triulzi. A SSH server in your NIC. PacSec 2008.
[245] Tudor, Valentin and Almgren, Magnus and Papatriantafilou, Marina. Analysis of the impact of data granularity on privacy for the smart grid. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, 2013.
[246] A. Valdes and K. Skinner. Adaptive, Model-Based Monitoring for Cyber Attack Detection. In *Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science. Springer, 2000.
[247] V. van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In *Proceedings of The 15th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'12, 2012.
[248] S. Wang, L. Cui, J. Que, D.-H. Choi, X. Jiang, S. Cheng, and L. Xie. A randomized response model for privacy preserving smart metering. *Smart Grid, IEEE Transactions on*, 3(3):1317 –1324, sept. 2012.
[249] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy*, pages 497–512, 2010.
[250] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, SP 10, 2010.
[251] S. D. Warren and L. D. Brandeis. The right to privacy. *Harvard law review*, 4(5):193–220, 1890.
[252] R.-P. Weinmann. Baseband attacks: remote exploitation of memory corruptions in cellular protocol stacks. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, WOOT'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
[253] H. Welte. Anatomy of Contemporary GSM Cellphone Hardware.
[254] D. A. Wheeler. SLOCCount – a set of tools for counting physical Source Lines of Code (SLOC). http://www.dwheeler.com/sloccount/.
[255] M. Williams. ARMV8 debug and trace architectures. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pages 1–6, 2012.
[256] N. Williams, B. Marre, and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE'04, 2004.
[257] xobs and bunnie. The Exploration and Exploitation of an SD Memory Card. *CCC – 30C3*, 2013.
[258] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.
[259] Yahoo S4. http://incubator.apache.org/s4/.
[260] Y. Yan, Y. Qian, and H. Sharif. A secure and reliable in-network collaborative communication scheme for advanced metering infrastructure in smart grid. In *Wireless Communications and Networking Conference (WCNC), 2011 IEEE*, pages 909 –914, march 2011.
[261] Y. Yan, Y. Qian, and H. Sharif. A secure data aggregation and dispatch scheme for home

area networks in smart grid. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1 –6, dec. 2011.

[262] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 21st Symposium on Network and Distributed System Security*, NDSS '14. The Internet Society, 2014.

[263] J. Zaddach and A. Costin. Embedded Devices Security and Firmware Reverse Engineering. *BlackHat USA*, 2013.

[264] J. Zaddach, A. Kurmus, D. Balzarotti, E. O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *ACSAC 2013, 29th Annual Computer Security Applications Conference, December 9-13, 2013, New Orleans, Louisiana, USA*, New orleans, UNITED STATES, 12 2013.

[265] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas. Implementation and Implications of a Stealth Hard-drive Backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 279–288, New York, NY, USA, 2013. ACM.

[266] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *Proc. of the 3rd International Conference on Software Testing, Verification and Validation*, ICST'10, Apr. 2010.

[267] D. Zinn, Q. Hart, T. McPhillips, B. Ludascher, Y. Simmhan, M. Giakkoupis, and V. K. Prasanna. Towards reliable, performant workflows for streaming-applications on cloud platforms. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011.

[268] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, Nov. 2004.