Convincing Without Revealing: Strategies for Facilitating Remote Attestation under Weakened Trust Assumptions using Privacy-Enhancing Technologies

Debes, Heini Bergsson

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

Convincing Without Revealing: Strategies for Facilitating Remote Attestation under Weakened Trust Assumptions using Privacy-Enhancing Technologies

Heini Bergsson Debes
PhD Thesis

December 31, 2022

Technical University of Denmark
Department of Applied Mathematics and Computer Science
ISSN: 0909-3192

Summary

Remote attestation is the process in which one computing system, known as the prover, supplies evidence about some claim to another computing system, known as the verifier, which may be located at a remote location. For example, in claiming to be in a correct software state, the prover might supply evidence in the form of a measurement of its current software, or in claiming that a specific trusted authority has given it certain rights, it might supply a cryptographic signature produced by that authority. It is a valuable tool for maintaining the trustworthiness and security of devices and systems in the digital age. However, one major challenge that needs to be addressed to encourage wider adoption of certain remote attestation protocols is privacy. Remote attestation often requires the exchange of sensitive information, which can compromise the privacy of the parties involved and discourage adoption in privacy-critical sectors.

This PhD thesis focuses on developing enhanced remote attestation protocols that address the privacy challenge inherent in remote attestation by leveraging Trusted Computing and Privacy-Enhancing Technologies (PETs). The main protocols include: a Trusted Platform Module (TPM)-based protocol that gives verifiers assurance of a platform's configuration integrity without revealing any platform information; a Control-Flow Attestation (CFA) protocol based on Verifiable Computation that enables even the most resource-constrained computing systems to prove the correct execution of security-critical programs without disclosing any program details; and a protocol that joins commit-carrying zkSNARKs and designated-verifier proofs with Anonymous Credentials as a more privacy-respecting and expressive alternative to traditional authentication protocols.

The aim of these enhanced remote attestation protocols is to encourage wider adoption of remote attestation in this connected world by providing a way for verifiers to reason about a remote prover in a privacy-respecting way without sacrificing security.

Summary (Danish)

Fjernattestering er processen, hvor et computingsystem, kendt som beviseren, leverer beviser om en påstand til et andet computingsystem, kendt som verificereren, som kan være placeret på et fjernsted. For eksempel kan beviseren, der påstår at være i en korrekt softwaretilstand, levere beviser i form af en måling af deres nuværende software, eller hvis de påstår at en specifik betroet autoritet har givet dem visse rettigheder, levere en kryptografisk signatur produceret af denne autoritet. Det er et værdifuldt værktøj til at opretholde troværdigheden og sikkerheden for enheder og systemer i den digitale tidsalder. En stor udfordring, der skal tages højde for for at opmuntre til bredere adoption af visse fjernattesteringsprotokoller, er privatlivet. Fjernattestering kræver ofte udveksling af følsomme oplysninger, hvilket kan udgøre en trussel mod privatlivet for de involverede parter og afskrække adoption inden for privatlivskritiske sektorer.

Denne PhD-afhandling fokuserer på udvikling af forbedrede fjernattesteringsprotokoller, der tager højde for privatlivstruslen, der er indbygget i fjernattestering ved at udnytte pålidelig computing og teknologier, der forbedrer privatlivet (PET'er). De vigtigste protokoller inkluderer: en protokol baseret på Trusted Platform Module (TPM), der giver verificererne sikkerhed for en platformkonfigurations integritet uden at afsløre nogen platformoplysninger; en Control-Flow Attestation (CFA) protokol baseret på verificérbar beregning, der gør det muligt for selv de mest ressourcebegrænsede computingsystemer at bevise korrekt udførelse af sikkerhedskritiske programmer uden at afsløre nogen programdetaljer; og en protokol, der sammenkæder commit-carrying zk-SNARKs og designated-verifier proofs med anonymous credentials som et mere privatlivsbevarende og udtryksfuldt alternativ til traditionelle autentiseringsprotokoller.

Formålet med disse forbedrede fjernattesteringsprotokoller er at opmuntre til bredere adoption af fjernattestering i denne forbundne verden ved at give verificererne mulighed for at evaluere en fjern beviser på en privatlivsbevarende måde uden at gå på kompromis med sikkerheden.

Preface

The work presented in this thesis was conducted in the Cybersecurity Engineering section at the Department of Applied Mathematics and Computer Science (DTU Compute) in partial fulfillment of the requirements of the PhD program. The research was funded by the European Union's Horizon 2020 research and innovation program under the RAINBOW project Grant Agreement 871403 (https://cordis.europa.eu/project/id/871403). It was carried out under the supervision of Christian D. Jensen and Thanassis Giannetsos in the period from January 2020 to December 2022.

Acknowledgements

I want to thank my supervisors, Christian D. Jensen and Thanassis Giannetsos, for their advice during my PhD studies. I would also like to thank my colleagues at the Cybersecurity Engineering section at the Technical University of Denmark (DTU) for their support and discussions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This introductory chapter gives some motivation and provides an overview of the main works done in this thesis. Let us begin.

The computing world is rapidly evolving, and the number of computing systems is increasing, ranging from powerful desktop computers to deeply embedded systems. With cloud computing, we can access data and applications from anywhere in the world, and with our increasingly resourceful mobile devices, we can stay connected and undisturbed. With the Internet of Things (IoT), our devices can communicate and exchange data with each other and the internet, leading to smart homes, connected cars, and industrial automation systems. With Cyber-Physical Systems (CPS) [134], the line between physical and digital is diminishing even further as we attempt to combine the remaining physical system infrastructure with the world of computing systems. These CPSs are complex systems that combine cyber (digital) and physical elements to perform tasks or achieve goals. Today, they are often found in critical infrastructure, at the core of healthcare devices, electrical power grids, weapons systems, transportation management systems, industrial control systems, and other sectors where the integration of computing and physical components is necessary to achieve some desired level of automation and control. Here the computing elements, such as computers and sensors, coordinate and communicate with physical components, such as actuators and other mechanical machinery.

Generally, the purpose of a CPS is to control some physical process and maintain it in some desired state. Such systems typically comprise a set of sensors, a controller, and some actuators. The sensors are tiny devices capable of detecting and measuring physical phenomena such as temperature, pressure, or motion and might be used to monitor the state of the physical process under control by the CPS. These sensor readings are then reported to the controller, which sends control signals to actuators (e.g., a valve) to maintain the system in some desired state by modifying the cyber and physical environment. For example, an actuator might be used to open or close a valve or move a robotic arm. The controller might also communicate with a supervisory or configuration device, such as a Supervisory Control and Data Acquisition (SCADA) system [6], which can monitor the system or change the controller's settings. For example, in a power grid, the SCADA system might be used to monitor and control the flow of electricity through the grid [53].

With the proliferation of such systems, the possibilities appear endless, and we are reaping the rewards with greater convenience, faster communication, and access to vast

information and resources. From controlling our home automation systems, including saving costs by dynamically adjusting light and heating levels according to various sensor readings and online intelligence, to better situational awareness control and increased precision for self-protection systems onboard military vessels based on the integration with sensor data, command and control (C2) systems, and high-precision actuators.

Nevertheless, despite its numerous benefits, there is no denying that this continuing technological revolution raises important questions regarding privacy, security, and the potential effects on society [79]. For example, as described in [53], industrial control systems [6] perform vital functions in critical national infrastructures, such as electric power distribution, oil and natural gas distribution, water and wastewater treatment, and intelligent transportation systems. The disruption of these CPSs can severely affect public health and safety and lead to economic losses. For example, attacks on power grids can cause blackouts, leading to cascading effects in other vital critical infrastructures. Attacks on ground vehicles can create highway accidents, attacks on GPS systems can mislead navigation systems, attacks on medical equipment can endanger human lives, and one can only imagine the consequences of successful attacks against weaponry systems.

Some problems are already clear. Our systems, from cloud services to deeply embedded CPSs, which often rely heavily on software-based automation, implicitly trust code developers to write perfect software that operates expectedly and cannot be compromised for malicious purposes [138]. However, this is not always the case, and malicious cyber actors often find and exploit software vulnerabilities to gain remote code execution capabilities or cause other adverse effects. For example, one of the most common software vulnerabilities is caused by insecure memory management, as corroborated by numbers from Microsoft in 2019 [133], which revealed that in the period between 2006 to 2018, approximately 70 percent of their vulnerabilities were due to memory safety issues. Such issues are found in software written in commonly used languages, such as C and C++, which provide a lot of freedom and flexibility for developers to maneuver memory but ultimately rely on the programmer to ensure that everything is done safely and that memory is properly allocated and deallocated. Here, simple mistakes can lead to exploitable memory-based vulnerabilities. For example, suppose the programmer forgets to check that a write to a data variable (e.g., a C array) does not exceed the size of the allocated memory buffer. In that case, the buffer can overflow by simply writing more data than it was originally meant to hold, forcing the excess data to overwrite adjacent memory, which might belong to other program variables or even state information used by the Central Processing Unit (CPU) to guide the program's execution. With such vulnerabilities, meticulous attackers can remotely crash or even alter the expected execution of the affected program by purposely overflowing the vulnerable buffer and injecting carefully constructed data or executable code snippets into the victim's memory.

Therefore, in a proactive attempt to detect potential software vulnerabilities when writing the software, developers might employ common code analysis tools and techniques to identify and address potential vulnerabilities. These tools and techniques may include static code analysis, which involves analyzing the program code carefully to find vulnerabilities without executing the code, and dynamic code analysis, which involves executing code and monitoring its behavior when given certain inputs. Other techniques for developers of security-critical software might also include fuzz testing, software penetration testing, and practicing general secure programming principles, including code reviews, following standardized coding rules, and performing thorough compliance test-

ing. However, even with such rigorous, proactive detection strategies to prepare the logic in software for surprising conditions, commonly exploitable software vulnerabilities, such as those based on memory issues, might still slip through the cracks. Thus, in an effort to "patch" commonly used programming languages that are susceptible to producing vulnerable code (e.g., C and C++), additional prevention strategies have been introduced, such as dedicated and secure libraries that offer code developers access to high-level APIs that take care of certain error-prone tasks, such as memory management, by providing the developer with abstractions such as fat pointers, garbage collection, and smart pointers. However, due to a continued risk of memory safety issues (among others) leading to exploitable software vulnerabilities such as buffer overflows and other wormable flaws, a shift has also recently begun towards more memory-safe languages (e.g., C#, Go, Java, Ruby, Rust, and Swift). This transition towards newer, safe-by-design programming languages is also urged by major software firms and national-level intelligence agencies, including the National Security Agency (NSA) [138].

However, even with good techniques and tools to help prevent introducing vulnerabilities in new code, they may not be complete, are not enforced by everyone, and cannot easily be applied to legacy code. Therefore, other complementary mitigation strategies are commonly added directly to the execution infrastructure, e.g., the hardware and operating system on which software is executed. Some common mitigations include data execution prevention (DEP), for preventing code injection by marking certain memory pages (e.g., the stack and heap) as *non-executable* during runtime, and obfuscation, such as applying fine-grained code randomization [122] or address space layout randomization (ASLR) [70, 168], for rearranging how programs are stored in memory to make it harder for attackers to plan out effective attacks. Other mitigations often focus on specific properties that are supposed to hold during a program's execution. For example, stack canaries [50] and shadow call stacks [74, 1] are often employed as mitigation against buffer-overflow and Return-Oriented Programming (ROP) attacks [162], often by guarding certain memory locations where return addresses are stored during a program's execution (see, e.g., [188]). Program functions use these addresses to remember whom to call after they are done executing and, if left unprotected, could be overwritten by malicious attackers, giving them complete control over the program's execution. Another well-known technique for protecting a program's execution is Control-Flow Integrity (CFI) [1]. The general idea here is to monitor the program and ensure it executes as expected. Given an allowlist or a reference copy of a program, CFI actively monitors a program during its execution and continuously verifies that the instructions and functions are executed as expected (e.g., in the correct order) according to the trusted reference material.

However, CFI and similar mitigation solutions may not be desirable in all systems. For example, CFI is an inherently program-specific solution that relies on program-specific trusted reference materials. Due to scarce resources and a small trusted computing base, such reference materials can be challenging to maintain with over-the-air software updates for a deeply embedded CPS. Another problem is that the runtime verification affects the program performance and the device's power consumption. Thirdly, the typical reaction when detecting a violation is to terminate the affected program, which might inadvertently affect the device's availability. This issue is of particular concern in certain safety-critical sectors, where device unresponsiveness might cause physical and safety hazards. Finally, note that CFI is generally intended as a purely local enforcement solution and, thus, provides no means for a device to report a program's correct execution

to remote devices. This is especially a problem if we are to rely on the data generated on a remote device. For example, consider distributed power grids [40], where local controllers might execute controls based on data generated on a sending device and received over communication channels. Here the receiving controller should be able to remotely verify the computational integrity of the control algorithm that produced the data before relying on it to guide its controls. Additionally, it might be important for the receiving controller to verify that the device that produced the data is authentic, has been properly configured, and is authorized to decide how the local controller should act. On the other hand, the sending device's control algorithm, configuration, and credentials might be confidential. Solving this double-edged issue is of primary concern in this thesis.

We can essentially consider each of these problems (i.e., proof of correct: program execution, configuration state, and credentials) as separate "proof statements" about which a prover (e.g., the confidential device) wishes to convince a verifier (e.g., the local controller) without having to reveal any sensitive information. The main contribution of this thesis includes three privacy-respecting protocols, each dedicated to solving a particular problem using state-of-the-art Trusted Computing practices and Privacy-Enhancing Technologies, as depicted in Fig. 1.1.

Figure 1.1: Problem statement and main idea. The big duckling (verifier) wants to inspect the little duckling (prover), but the little duckling feels exposed. With Trusted Computing and Privacy-Enhancing Technologies, verifiers can reliably inspect provers without infringing on their privacy (depicted as obscured inspection glass). Credit to DALL·E 2.

1.1 Summary of Main Contributions

Concerning the first problem, we consider the remote variant of CFI, called Control-Flow Attestation (CFA) [2, 63, 186, 62, 3, 176, 95, 125, 169, 117, 140, 141, 187], where the idea is for the device that executes the program (called the prover) to outsource the verification to a remote device (called the verifier). Here, the prover does not need to spend additional resources to perform the verification locally, nor does it have to maintain up-to-date versions of all trusted reference materials. In CFA, the prover is only tasked with recording the program's execution trace, signing the trace using a cryptographic key, and sending

the signed trace to the verifier, who then performs the necessary verification. While CFA schemes differ in certain aspects, such as how tracing is performed and how the trace is structured, they generally follow this general idea on the surface. However, an issue with existing CFA schemes is that they do not offer any privacy guarantees. For example, suppose a device executes a confidential program (e.g., the control algorithm mentioned above) or wants to keep sensitive program details hidden from potentially dishonest verifiers. In that case, existing CFA schemes cannot be used since verifiers would need access to the sensitive reference materials to perform the verification. To solve this issue, we proposed a novel scheme called ZEKRA that uses privacy-preserving Verifiable Computation based on zkSNARKs [84, 114] to transform the CFA verification algorithm into an outsourceable circuit for checking the correctness of a program's execution trace according to the program's reference Control-Flow Graph (CFG). The idea is that since the CFG models all legal paths in a program, we know that the program's trace is free of control-flow-based runtime attacks (e.g., Return-Oriented Programming attacks) if it is correct according to the CFG. Then, in collaboration with a worker device, the prover can use the circuit to create a zero-knowledge proof about the correctness of the recorded program's execution trace. Any verifier can then verify the proof without needing access to any sensitive reference materials to check whether the program executed correctly and in the absence of runtime attacks.

Similar to the idea behind CFA, remote attestation [7, 184, 156, 126, 127] has become increasingly popular as a Trusted Computing method for reliably verifying the correctness of devices in remote settings. As with CFA, it generally assumes that devices have some form of a trusted computing base capable of securely measuring the device's state, e.g., taking a snapshot of the memory, the filesystem, or the firmware's configuration and signing the measurement using a securely kept cryptographic key. The signed measurement can then be reported to remote verifiers, who might compare it against trusted reference values to determine if it is trusted. See, e.g., [10] for an overview of existing remote attestation schemes and techniques. However, note that the use of remote attestation also raises privacy concerns, like that of CFA. Specifically, it requires the device to collect and report sensitive information, such as its hardware and software configuration, which can foster discrimination attacks and provides valuable profiling intelligence to malicious actors planning to launch attacks against a system. It is, therefore, essential to ensure that remote attestation is carried out in a secure and privacy-preserving manner. While some remote attestation schemes offer different forms of privacy [185, 45, 9, 7, 184, 126, 127], they tend to limit verifiability or introduce additional assumptions on the network setup. We propose a novel scheme called ZEKRO [60] that uses the enhanced authorization functionality in the Trusted Computing Module (TPM) [172] to allow a prover to continuously convince verifiers about its configuration integrity correctness (e.g., that its filesystem or memory is in a correct state) without disclosing any concrete information.

Concerning the third problem, we consider anonymous credentials [14, 32, 90, 33, 31], which is another increasingly popular technique for users and other entities (e.g., devices) to hold cryptographic evidence about their identity and properties (e.g., a digitally signed diploma, driver's license, or bank statement). After being issued a credential from a trusted authority (e.g., a governmental agency), the credential holder can prove to others that it indeed possesses a valid credential and may also selectively reveal attributes in the credential. For example, a person might reveal the age attribute in their digital passport to enter a bar, or the confidential device in the power grid example above might re-

veal the attributes specifying its role and cell (domain) to prove authorization to the local controller. In this direction, we propose a novel, fully expressive, and designated-verifier anonymous credentials scheme called RETRACT. The scheme utilizes Verifiable Computation based specifically on commit-carrying zkSNARKs [37] to facilitate expressiveness (i.e., allow a credential holder to prove arbitrary criteria about its credentials), BBS+ signatures [31] to create the anonymous credentials, and trapdoors to make all proofs designated-verifier [104] (i.e., ensuring that only a particular verifier can use the proof). The developed scheme allows credential holders to prove, in a privacy-respecting manner, possession of credentials and arbitrary statements about the credential attributes. The scheme also ensures that only the intended verifier is convinced by a proof.

1.2 Publications

The set of publications which are included in this thesis is listed below:

A **Heini Bergsson Debes** and Thanassis Giannetsos. 2021. Segregating Keys from *nonce*nse: Timely Exfil of Ephemeral Keys from Embedded Systems. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE.

B **Heini Bergsson Debes** and Thanassis Giannetsos. 2022. ZEKRO: Zero-Knowledge Proof of Integrity Conformance. *2022 17th International Conference on Availability, Reliability and Security (ARES)*. 1-10.

C **Heini Bergsson Debes**, Edlira Dushku, Thanassis Giannetsos, and Ali Marandi. 2023. ZEKRA: Zero-Knowledge Control-Flow Attestation. To appear in *2023 18th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS)*.

D **Heini Bergsson Debes** and Thanassis Giannetsos. RETRACT: Expressive Designated Verifier Anonymous Credentials. (Under submission.)

Note that while Section 1.1 introduced the problems of proving correct program execution (C), configuration state (B), and credentials (D) in an intuitive order, here we have listed the publications in chronological order as they were produced (B, C, D). Additionally, note that a fourth paper (A) was not directly introduced in Section 1.1. Before investigating privacy-preserving security protocols aimed at detecting sophisticated attacks and enhancing trust between mutually distrusting parties, this initial paper started at the other side of the spectrum. Specifically, in this first paper [59], we investigated how an advanced attacker can remotely infect resource-constrained embedded devices (in our case, the Tmote Sky module) and systematically exfiltrate data likely to contain cryptographic keys used by deterministic system routines (e.g., reception handlers) from the memory stack without having any knowledge about the software. We also demonstrated how the attacker could apply specific data mining approaches on the exfiltrated data to reasonably reduce the search space, thus making the hunt for the cryptographic key more efficient. While a different focus from the remaining contributions, we can essentially think of this paper as a nice "motivation", especially for the two succeeding works (B and C) that investigated remote attestation schemes to detect such attacks.

Besides these four main works, the publications listed below were also produced during the PhD study. The first (paper E) focused on using the enhanced authorization functionality of the TPM to create policies that restrict a cryptographic key's use to the state of a Platform Configuration Register (PCR), thus preventing a prover from producing valid

signatures unless it is in a correct state. Furthermore, since the number of such PCRs is scarce on a TPM, the second paper (F) showed how to additionally utilize the TPM's non-volatile memory for holding more PCRs. This additional flexibility boosts privacy in multi-tenant cloud environments by giving each tenant their exclusive PCR instead of sharing. However, since these works are subsumed in the ZEKRO [60] protocol (paper B), except for some technical differences, they are not included in this thesis.

E Benjamin Larsen, **Heini Bergsson Debes**, and Thanassis Giannetsos. 2020. CloudVaults: Integrating Trust Extensions into System Integrity Verification for Cloud-based Environments. In *2020 25th European Symposium on Research in Computer Security (ESORICS)*. Springer, 197-220.

F **Heini Bergsson Debes**, Thanassis Giannetsos, and Ioannis Krontiris. 2021. BlindTrust: Oblivious Remote Attestation for Secure Service Function Chains. *arXiv preprint arXiv:2107.05054 (2021)*. (In progress.)

1.3 Outline

Besides this introductory chapter, this thesis is organized into four main branches, each dedicated to one of the main publications produced during the PhD study. The chapters appear in the order in which the publications were made, as depicted in Fig. 1.2. Specifically, we begin with the "motivational" paper in Chapter 2, then Chapter 3 gives ZEKRO, Chapter 4 gives ZEKRA, and finally, Chapter 5 gives RETRACT. Each of these chapters has a similar format, beginning with an introduction to the topic, followed by a summary of the specific contributions, the related works, a description of the background, the results and evaluation, and a conclusion. Finally, Chapter 6 concludes the thesis.

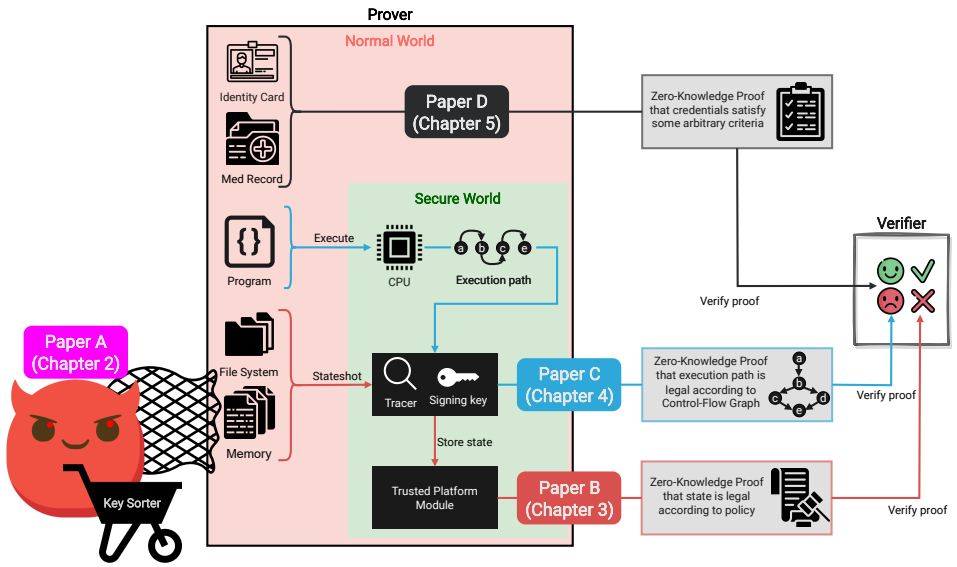


Figure 1.2: Thesis overview and outline.

Chapter 2

Segregating Keys from *nonce*nse: Timely Exfil of Ephemeral Keys from Embedded Systems

Abstract

As lightweight embedded devices become increasingly ubiquitous and connected, they present a disturbing target for adversaries circumventing the gates of cryptography. We consider the challenge of exfiltrating and locating cryptographic keys from the run-time environment of software-based services when their software layout and data structures in memory are unknown. We detail an attack that can, without affecting the target system's operation, systematically exfiltrate certain keys during their use by leveraging the strong causality between transceivers and keyed cryptosystems (authentication, authorization, and encryption). We then propose how to effectively and efficiently reduce the key material's search space from a batch of stackshots (stack extractions) by leveraging the stack's innate composition, which, to the best of our knowledge, is the first method to systematically infer and reduce the search space of semi-arbitrary keys. We instantiate and evaluate our attack against MSP430 micro-controllers, which is one of the most prominent embedded systems used in many IoT-based applications.

2.1 Introduction

With the advent of the Internet of Things (IoT), deployment of embedded systems has accelerated, and systems have become unprecedentedly network-connected, autonomous, and collaborative (e.g., smart homes, automotive, healthcare, agriculture, industrial). However, in current security, privacy, and safety-critical IoT application domains, a significant portion of *resource-constrained* microcontroller-based embedded systems (MCUS) have

the inherent lack, due to cost, of tamper-proof hardware or essential protection mechanisms found in their desktop counterparts, e.g., cryptographic modules such as Trusted Platform Modules (TPM), Data Execution Prevention (DEP), stack canaries, and Address Space Layout Randomization (ASLR) [115, 8], thus leaving MCUS susceptible to an increasing number of remote attacks [142], such as the remotely-exploitable vulnerabilities in the RTL8195A module [142].

Coupled with resource-constrained Operating Systems (OS, e.g., Contiki, FreeRTOS, or TinyOS [175]) or bare-metal (without any OS), MCUS run single binary images where the application orchestrates all system resources and, due to the lack of onboard protection mechanisms, determines the system's security posture, which, generally comprises keyed authentication, authorization, or encryption cryptosystems. However, such cryptosystems' security lies solely in the secrecy of the *secret* key (Kerckhoffs's principle); so does that of MCUS.

To retain key secrecy in *all* stages (creation, dissemination, storage, and *usage*) is non-trivial. Computing systems inherently require that any program, including its data and instructions, be loaded into the main memory before being run by the processor. Thus, while keys can be protected while stored [38], any conventional program that performs keyed cryptographic operations must, at some point, have the key material exposed [105], which is known as the Key Exposure Problem (KEP). Note that the KEP might not be a problem if we assume that key management is only performed behind closed curtains. However, since adversaries are indeed able to infiltrate MCUS, we must opt to consider adversaries peeking behind said curtain.

While there already exist ways to get a peek [163, 88, 76], the inherent deficiency lies in assuming that keys are visible in memory at acquisition time. Until now, attempts to narrow on the timeliness [171, 13] or locating (or reduce the search space of) key material once the memory is obtained [163, 109, 150, 88, 92] have been limited to specific cryptosystems or software implementations.

2.1.1 Contributions

We demonstrate how adversaries can, *without knowing the software layout or memory data structures of running services*, exploit the KEP in *network-connected* MCUS to exfiltrate cryptographic keys, during system operation systematically, *without* affecting system usability. We present *generic* methods for overcoming two significant challenges revolving around *successful* key exfiltration: (i) how to acquire the memory contents systematically *while* the key is exposed (exfiltration phase) and (ii) how to efficiently reduce the search space of arbitrary key material (localization phase). Specifically, targeting the nature of MCUS, we demonstrate how to exploit the causality between transceiver invocation and utilization of keyed cryptosystems to acquire timely memory extractions. Concretely, since keyed cryptosystems inherently run post-reception (e.g., to verify or decrypt an incoming payload), we can, by periodically exfiltrating conventionally used memory regions for storing run-time data (e.g., the memory *stack*), capture data belonging to the keyed cryptographic function during the inevitable Key Exposure Window (KEW) caused by the KEP. Further, as a software- and cryptosystem-*agnostic* method of locating key material, we propose to apply specific data mining techniques (Section 2.5.5) that leverage the stack's innate composition. The intuition behind the presented work is to showcase how vulnerable the existing commodity MCUS are against sophisticated

attacks and emphasize the need for appropriate prevention strategies (Section 2.8). To encourage further work, we also make our prototype code publicly available [55].

2.2 Related Work: Toward Key Identification

In 1998, after observing the inherent randomness in cryptographic keys, Shamir et al. [163] postulated that memory regions with unusually high entropy might infer a key's presence. However, their conjecture that keys have higher entropy than other data is not always valid [88]. The approach becomes even less attractive when considering symmetric keys, which are conventionally much smaller than asymmetric keys. For symmetric keys, Halderman et al. [88] proposed searching for mathematical properties of the AES key schedule and further conceptualized a semi-unified search for different types of keys (symmetric and asymmetric) by incorporating heuristics about the cryptographic algorithms, e.g., well-known RSA encodings wherein the key is encapsulated within fixed structures [109] or the memory reflections of code structures containing key material in standardized implementations [150]. The inherent deficiency of each approach is that it is viable only for a target algorithm or is applicable *only when implementations yield the presumed structural representation in memory*. For example, although the AES key schedule has distinctive characteristics, it cannot be said about symmetric keys in general as randomness is their only definite identifiable characteristic. Further, the schedule's structural reflection in memory is also *not* fixed *nor* certain. Such issues make any attempt toward unified key localization from memory contents *difficult*.

To narrow the search space on which key identification is conducted, the authors of [92] propose reconstructing call stacks of functions that invoke security-sensitive Windows APIs, known to accept keys as arguments. The premise (which is also used in this paper) is that program functions generally use the memory stack to hold variables during execution, and thus the stack of keyed functions will inevitably contain key materials. However, besides being highly application dependent, in practice, compiler optimizations will severely complicate call stack reconstruction (e.g., its memory reflection and which elements occur) [167, 41]. Further, as they note, there is no guarantee that a key is in memory at the acquisition time.

Considering *timeliness*, authors of [13] target Ransomware keys by monitoring invocations of specific cryptographic APIs to *trigger* memory extraction and authors of [171] monitor the control-flow of network-related functions in Android applications to *trigger* acquisition of TLS key materials. However, whereas they require direct control-flow monitoring capabilities, we propose a more lightweight and application-*agnostic* trigger heuristic. Further, where they employ algorithm- and implementation-aware key identification techniques, we propose novel exploitation of the program stack's nature to isolate areas likely to contain *whichever* key that might be present.

2.3 Problem Statement

The strength of a cryptosystem, with key-length kl, is quantified by its ability to resist brute-force attacks on the entire key-space $2^{|kl|}$, which *should* be computationally infeasible for large kl's. However, applying an exhaustive search on a considerably *reduced*

search space of memory is appealing and, in theory, more efficient, as the search space reflects all possible key-sized blocks of contiguous bytes. Let D denote a device with volatile memory VM. The shared memory space is defined as $SM = \{s, r, h\} \in VM$, comprising the stack s, registers r, and the heap h, respectively. Let P be a program running on D which uses SM as its execution environment. For simplicity, assume that $SM_t = \langle s_t \| r_t \| h_t \rangle$ is the finite sequence of bytes stored in SM at time t and $|SM_t|$ denotes the cardinality. Let $\mathcal{SS} = \{SM_t : t \in T \subset \mathbb{N}^*\}$ denote the search space comprising all instances of SM in D's universal time space $\mathcal{T}$. Further, let $\mathcal{K}$ be a key used by P such that $\mathcal{K}$ is stored in its *entirety somewhere* in SM at times $T^{\mathcal{K}} \subseteq \mathcal{T}$.

Let $\mathsf{Ext} : T \times \mathbb{N}^* \times \mathbb{N}^* \rightarrow \mathcal{SS}$ be a three-input extractor function accepting a time t, a start index i, and a range rg to return a subsequence from SM at time t. It is defined by $\mathsf{Ext}(t, i, rg) = \langle a_i, a_{i+1}, \ldots, a_{i+rg} \rangle \sqsubseteq SM_t$, where $1 \leq i < i + rg \leq |SM_t|$ and $\sqsubseteq$ denotes *subsequence of*. Hence, $\forall t \in \mathcal{T}$ we define a map instance $\mathsf{Ext}_t : \mathbb{N}^* \times \mathbb{N}^* \rightarrow \mathcal{SS}$ by $\mathsf{Ext}_t(i, rg) = \mathsf{Ext}(t, \langle a_i, a_{i+1}, \ldots, a_{i+rg} \rangle)$.

We formalize the search space reduction ($\mathcal{SSR}$), where we consider an adversary ($\mathcal{A}$) that, based on select subsequences from instances of Ext, tries to find the fewest possible candidates for $\mathcal{K}$. The $\mathcal{SSR}$-complexity is defined as the number of candidates and is over the choices of t, rg, and any choice of $\mathcal{A}$ herself. Accordingly, $\mathcal{A}$ is given oracle access to Ext so she can obtain subsequences of her choice and is not constrained concerning the method she uses, leading to Definition 2.3.1.

Definition 2.3.1. Let B be a $\mathcal{SSR}$ algorithm that takes the function map Ext and yields a set of subsequences (*candidate keys*). Considering the experiment in Algorithm 1, the *$\mathcal{SSR}$-complexity* of B is defined as: $\mathsf{CXTY}^{\mathrm{ssr}}_{\mathsf{Ext}}(B) = \mathsf{Exp}^{\mathrm{ssr}}_{\mathsf{Ext}}(B)$.

Algorithm 1: Experiment $\mathsf{Exp}^{\mathrm{ssr}}_{\mathsf{Ext}}(B)$

Input : $\mathcal{SSR}$ algorithm, B
Output: Cardinality of the set of candidate keys
1 $\sigma \leftarrow B^{\mathsf{Ext}}$
2 **if** $\mathcal{K} \in \sigma$ **then return** $|\sigma|$ **else return** $\perp$

The definition is made general enough to capture all types of key-localization attacks. For example, performing a Variable Sliding-Window (VSW, i.e., *linear scan* [91]) attack over the entire contents in $\mathcal{SS}$, using window sizes $n = 1, \ldots, N \in \mathbb{N}^*$, yields an upper bound of: $\mathsf{CXTY}^{\mathrm{ssr}}_{\mathsf{Ext}}(VSW) = \sum_{n=1}^{N} |SM| - n + 1, \forall SM \in \mathcal{SS}$. This motivates our question: can $\mathcal{A}$ reduce the search space more efficiently (i.e., reduce the number of candidate keys) by applying heuristics such as data mining, logical reasoning, and inference? However, note that because the heap is rarely used in resource-constrained MCUS [78], it is not considered. Also, although registers generally contain valuable information, we argue that they are less likely to contain keys in resource-constrained MCUS (Section 2.5.3). Therefore, for this paper, the stack is of *exclusive* interest.

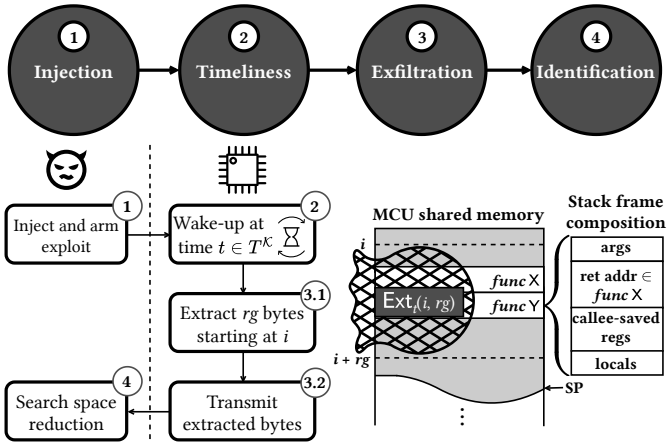


Figure 2.1: The four fundamental phases of key acquisition.

2.4 System and Adversarial Model

We consider how an *software-oblivious* $\mathcal{A}$ can methodically acquire ephemeral cryptographic keys (i.e., keys that are non-existent before use and securely sanitized immediately afterward) used for authentication, authorization, and encryption in remote resource-constrained MCUS, following the four attack phases depicted in Fig. 2.1. We assume that $\mathcal{A}$ can: (i) access the shared memory, (ii) periodically transmit select shared memory back to herself, and (iii) validate guesses for arbitrary $\mathcal{K}$. Note that while acquiring remote code execution is complementary to our work, we stress that $\mathcal{A}$ can achieve this advantage through a wide set of attack vectors, e.g., through code injection [78] or Return-Oriented Programming (ROP) [162] by exploiting software vulnerabilities [142]. As a running example, we consider the vulnerable reception handler in Fig. 2.4, where the infamous C *strcpy* function is exploited to unboundedly overwrite a *stack resident* buffer (the variable *c*), causing the stack frame's return address to point to malicious code (malcode) controlled by $\mathcal{A}$ as described in Section 2.5.4. Finally, although the software running on the device is a black box to $\mathcal{A}$ (compelling $\mathcal{A}$ to resort to *software-oblivious-centric* approaches that do not necessitate the source code), she is given knowledge about the target system's specifications, enabling $\mathcal{A}$ to leverage publicly accessible documentation. As a case study, we consider the prominent MoteIV Tmote Sky module [49] as our target.

2.4.1 Anatomy of the MoteIV Tmote Sky target

The Tmote Sky module is a TelosB sensor mote with an MSP430-F1611 microcontroller unit (MCU) [102] featuring a 16-bit CPU, and an CC2420 transceiver. The MCU components are interconnected following the Von Neumann architecture (i.e., where no hardware barriers prohibit the execution of injected code), typical for most general-purpose embedded systems, and employs the Memory-Mapped I/O (MMIO) paradigm for bilateral peripheral communication.

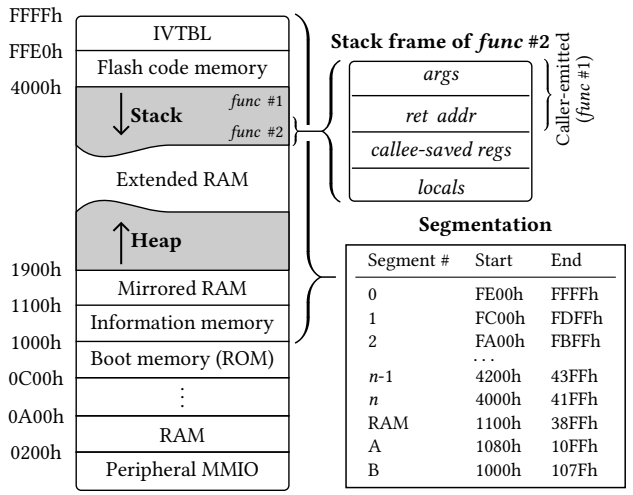


Figure 2.2: MSP430-F1611 memory and stack composition.

The MSP430-F1611 MCU's CPU [102] has a maximum clock frequency of 8 MHz. However, because the Tmote Sky module is a TelosB sensor mote, the CPU is restricted to 4 MHz, corresponding to $\approx 4\,000\,000$ Cycles Per Second (CPS). The CPU has 16 registers of 16 bits, numbered from R0 to R15. Four of the registers, R0 to R3, are dedicated as the Program Counter (PC), the Stack Pointer (SP), Status Register (SR), and Constant Generator (CG), respectively. The remaining 12 are general-purpose registers.

Figure 2.2 presents the memory map, comprising one address space shared with peripherals, RAM, and flash code. The extended RAM includes two oppositely growing data structures: the *stack* (discussed in Section 2.5.1) and *heap*. The stack is responsible for storing data during program execution, and the heap usually contains dynamically allocated memory. Flash code memory holds the executable program code and any constant data. Finally, for every hardware interrupt, the interrupt vector table (IVTBL) contains the 16-bit address of the appropriate Interrupt Service Routine (ISR).

2.4.1.1 Target Operating System

Many operating systems exist for cyber-physical systems besides TinyOS, such as MANTIS, Contiki, and LiteOS, to name a few. We consider TinyOS [175] as our target operating system, which is a simple C-based OS, as is valid for the majority of embedded systems. It is minimalistic, has a small RAM footprint of 400 bytes, and is programmed using a C-based dialect called nesC. It does not support dynamic memory allocation and, therefore, the heap (see Fig. 2.2) goes unused (unsupervised). Furthermore, TinyOS is single-threaded (single execution context) and does not have any preemption. Therefore, while any synchronous code is executing, it cannot be preempted by other synchronous code. However, the execution model is also event-driven, where low-level hardware interrupts (events) enforce task preemption. When an interrupt occurs, the processor immediately jumps to the handling code, i.e., the ISR stored in the IVTBL corresponding to the interrupt source.

2.5 Preliminaries and Methodology

2.5.1 Dissecting The Stack's Sequential Locality

The stack is a deterministic and sequential data structure. The structure is an aggregate of several consecutively allocated stack frames. Each function in a program is appointed one frame, serving as its scratch space and exists exclusively while the function is executing. The stack frame emulates a data structure (as depicted in Fig. 2.1) wherein the function organizes its local variables and other temporaries. As functions add (push) and remove (pop) stack elements during run-time, the Stack Pointer (SP) continuously points to the stack's top (most recent element). Although specifics (e.g., order) of specific frame elements may differ among architectures, the stack frames' structural reflection is always prevalent on the stack. When one function calls another, the caller must store a return address on the stack, pointing to an instruction that effectively resumes its halted execution once the callee finishes. The function prologue in the callee then stores specific volatile registers (states), which must be restored by the function epilogue before returning. Subsequently, the callee is free to utilize the remainder of its allocated frame for designated variables and temporaries (locals). Given these inherent characteristics, nested function calls inevitably leave a continuous trace corresponding to the call chain's depth and order. Any trail on the chain is guaranteed to endure until all of its descendants return; trails at the origin linger until the chain becomes completely unraveled. Thus, computationally costly functions (e.g., cryptographic operations) earn the key-exposure enablers' ribbon as they force their predecessors to persist and their locals to remain exposed on the stack.

Note that objects (arguments or locals) are eligible to be stored in a consecutive subset of available general-purpose registers, sequentially on the stack, or jointly by both. Although the compiler essentially decides which storage area to use, it must obey the target architecture's restrictions. For example, argument placement will inevitably depend on the argument's use (most registers do not support specific recursive operations), size, and registers' availability.

2.5.2 Inferring the Imminence of Key Exposure

Oblivious of when keys are used, $\mathcal{A}$ would have to consider all times as equally likely. However, since the predominant function of *network-connected* MCUS is data *transmission* and *reception*, some software function will inevitably be invoked to process incoming transmissions (e.g., to decrypt and verify). Thus, if $\mathcal{A}$ exploits the reception event to trigger $\mathcal{A}$-controlled code, $\mathcal{A}$ can effectively align stackshots to the inevitable KEW.

Since reception involves a transceiver peripheral, $\mathcal{A}$ can, in a *semi-software-oblivious* manner, exploit it. Specifically, transceivers (as other peripherals) will send signals on designated MCU pins to interrupt the application Central Processing Unit (CPU) about events, where, according to an Interrupt Vector Table (IVTBL), an Interrupt Service Routine (ISR) will run (e.g., to maneuver incoming data into application memory). Thus, given the MCU specifications, $\mathcal{A}$ can identify which IVTBL entry holds the reception ISR's memory address.

Note that despite the ability to locate an ISR's starting point, it is improbable to locate and traverse call-chains leading up to OS functions in a *software-oblivious* manner. Specif-

ically, while $\mathcal{A}$ can locate the reception ISR's starting address which causes invocation of the *receive* function in Fig. 2.4, the concrete ISR offset pointing to the call-chain entry remains unknown, prohibiting $\mathcal{A}$ from systematically attacking the function. Note, however, if $\mathcal{A}$ was unbounded, she could flush out the entire memory, attempt to deobfuscate the executable code, and then devise a software-dependent attack to acquire the key material post-reconstruction. However, since we aim for a stealthy, generic, and systematic method, such an attack is not in $\mathcal{A}$'s favor. Nevertheless, if $\mathcal{A}$ inserts a callback to commence periodic stackshots into the deterministically located reception ISR's prologue, stackshots will inevitably occur close to the KEW, assuming the causality between reception and keyed cryptosystems holds, *regardless* of the software underpinnings. Additionally, since most MCUS use the Memory-Mapped I/O (MMIO) paradigm for bilateral peripheral communication, another approach to align stackshots with the KEW is to identify and poll memory locations reflecting peripheral states. We consider both approaches to achieve timeliness in Section 2.6.

2.5.3 Inferring the Key's Presence

We proceed to infer the whereabouts of cryptographic keys in MCUS. Although data can theoretically reside anywhere (i.e., CPU registers, stack, heap, or other addressable memory regions), in practice, its placement is constrained by several factors, e.g., system resources, compiler, and adherence to the system's Application Binary Interface (ABI). As a running example, consider a reception handler that accepts and applies some arbitrary cryptographic function $\mathcal{F}$ on a packet's contents c using the key $\mathcal{K}$ (materialized using *reconstruct*). The general uses of $\mathcal{K}$, which affect its memory placement, are illustrated in Fig. 2.3 and include: $\mathcal{K}$ is declared and initialized as a function variable and passed by reference to $\mathcal{F}$ (use case #1); $\mathcal{K}$ is passed by value to $\mathcal{F}$ (use case #2); $\mathcal{K}$ is declared *above* function level and passed by reference to $\mathcal{F}$ (use case #3).

Use Case #1

```
void recv(uint8_t*c){
  uint8_t K[16];
  reconstruct(K);
  F(K,c); }
```

Use Case #2

```
struct S{
  uint8_t K[16]; }
void recv(uint8_t*c){
  F(reconstruct(),c); }
```

Use Case #3

```
void recv(uint8_t*c){
  reconstruct(K);
  F(K,c); }
```

Figure 2.3: The key ($\mathcal{K}$) is generally stored on the stack when it is a local variable or passed by value (#1 and #2) but not when it is a global variable and passed by reference (#3).

Since general-purpose CPU registers are inherently scarce in MCUS, keys are rarely eligible to be confined into registers. For example, the MSP430-F1611 has only 12 16-bit registers and prohibits objects exceeding 64 bits from occupying any registers [102]. Thus, considering the relatively small 16-byte $\mathcal{K}$ (Fig. 2.3), it inevitably resides on the stack in use cases #1 and #2. However, because $\mathcal{K}$ is declared *above the function level* in use case #3, $\mathcal{K}$ does *not* occur explicitly on the stack. Instead, $\mathcal{K}$ is put together with similarly scoped variables in a separate RAM region while its *reference* is passed to $\mathcal{F}$, either in registers or on the stack. Note, however, that although we target the stack

(Section 2.3), $\mathcal{A}$ could mitigate the uncertainty revolving around explicit and referenced keys by regarding each word on the stack as a potential reference and substitute it with a portion from the referenced memory if it references addressable memory. However, as words of a key might also resolve to addressable memory, additional care must be done. Nonetheless, for brevity, we assume that keys occur explicitly.

Despite $\mathcal{A}$'s inability to know $\mathcal{K}$'s definite stack placement, she can approximate it. Since cryptographic functions are inherently designed as leaf functions to mitigate key propagation issues [41], $\mathcal{K}$ likely remains within proximity of the SP during $\mathcal{F}$'s execution. For example, in use cases #1 and #2, $\mathcal{K}$ occurs in either the current ($\mathcal{F}$) or preceding (*recv*) stack frame. Thus, regarding effective use of Ext (Section 2.3), $\mathcal{A}$ has, besides *timeliness* (Section 2.5.2), a start index (SP), and justification that a short *range* (*rg*) can suffice for small cryptographic keys.

2.5.4 Exploiting the Stack Towards Arbitrary Code Injection and Stack Extraction

To remotely acquire the memory stack, $\mathcal{A}$ must first manage to inject the prerequisite code. The most prominent code injection method is to exploit neglected software vulnerabilities, such as buffer overflows, which enable sequences of code instructions to be injected and stored in contiguous memory regions chosen by $\mathcal{A}$ [78]. For TinyOS, because it is programmed using a C-based dialect, it inherits C's traits, including the absence of code safety. Therefore, as a running example, let us consider the buffer-overflow attack illustrated in Fig. 2.4 and our knowledge of the addresses in Table 2.1, where the infamous C *strcpy* function is exploited to unboundedly overwrite a *stack resident* buffer (the variable *c*), causing the stack frame's return address to point to malicious code (malcode) controlled by $\mathcal{A}$. For brevity, assume that the demonstrative reception handler has a stack frame (in practice, it might be inlined), and $\mathcal{A}$ knows the dedicated (alas, application-specific) memory buffer used to store received packets (Section 2.5.2) from which $\mathcal{A}$ can deduce the payload offset ($\text{ADDR}_{\text{payload}}$) by subtracting the packet's header length (controlled by $\mathcal{A}$). Also, assuming that $\mathcal{A}$ has managed to determine *where* the actual *return address* occurs on the stack, $\mathcal{A}$ can craft a malicious packet containing: sufficient bytes to overwrite values on the stack *up to* the return address ($\text{ADDR}_{\text{caller}}$); the substitute return address $\text{ADDR}_{\text{payload}}$ pointing within the malicious packet where malicious machine instructions (to be injected) are stored; a sequence of instructions which (when executed) copy some hard-coded bytes into a target memory region (ADDR_{TR}) chosen by $\mathcal{A}$; and finally, an instruction to restore normal control-flow ($\text{ADDR}_{\text{restore}}$). Hence, when *strcpy* is invoked (Step 1), the packet effectively replaces $\text{ADDR}_{\text{caller}}$ ($q_{\blacktriangleleft}$) with $\text{ADDR}_{\text{payload}}$ ($q_{\times}$, Steps 2 and 3), forcing the control-flow (PC) to $q_{\times}$ once the reception handler's function returns (Step 4), where the copy instructions contained in the packet's payload (Step 5) will be executed. Further, because the reception handler's normal execution flow starts at $q_{\blacktriangleright}$ (its prologue) and traverses through intermediate nodes until eventually passing q_i to reach $q_{\blacktriangleleft}$, where the caller is back in control, a prudent $\mathcal{A}$ must finish by joining the altered execution path with $q_{\blacktriangleleft}$, e.g., by artificially reverting the control-flow to $q_{\blacktriangleleft}$ ($\text{ADDR}_{\text{restore}}$, Step 6). Note that steps 1 through 6 repeat k times, where k denotes the number of packets required to inject the entire malcode. The dormant malcode is then triggered by redirecting the control-flow to ADDR_{TR} (Steps 7 and 8).

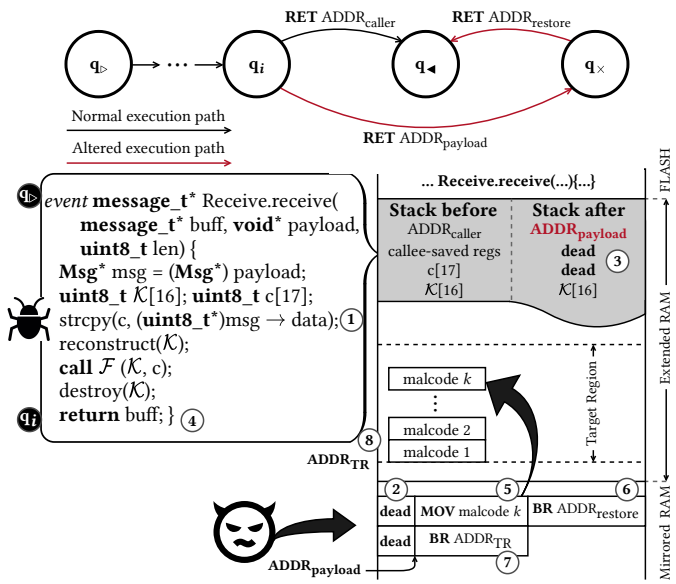


Figure 2.4: Multistage code injection through buffer-overflow.

Table 2.1: Requisite memory addresses for multi-staged code injection.

Address	Description
ADDR_{TR}	Unoccupied memory region targeted by $\mathcal{A}$
$\text{ADDR}_{\text{payload}}$	Address of the payload in a received packet
$\text{ADDR}_{\text{restore}}$	Original return address in reception handler's stack frame to restore control

2.5.5 Sequentially Mining Towards Search Space Reduction

To summarize our methodology so far. Section 2.5.2 motivated that we should attempt to align the memory extraction close to the reception of messages since a cryptographic routine might be invoked to handle the message. Section 2.5.3 proceeded to justify that it is reasonable to only consider the program stack, since the cryptographic key is likely to be stored on the stack temporarily when the supposed cryptographic routine is executing. Then, Section 2.5.4 presented a methodology for injecting our attack into the target device's memory.

The final challenge, and the question in our problem statement in Section 2.3, is how we can efficiently locate the supposed cryptographic key in the extracted data. Specifically, since the adversary only knows *when* an invocation of a reception handler invocation is *imminent* as described in Section 2.5.2, she must overestimate when the *key exposure window* (KEW) begins and take stackshots intermittently to ensure some overlap, which will result in an unwieldy growing search space. Indeed, for cryptographic keys with identifiable traits (e.g., entropy, statistical, structural, and mathematical), identification would be trivial. However, this is generally *false*, especially for symmetric keys

(Section 2.2), and assumes that $\mathcal{A}$ is aware of the type of key. Nonetheless, instead of resorting to an exhaustive search over the entire accumulated search space, $\mathcal{A}$ continues to devise an *efficient* $\mathcal{SSR}$ algorithm (by Definition 2.3.1) to keep the remaining $\mathcal{K}$-hammering effort within feasible levels. To $\mathcal{A}$'s advantage, although keys can take many forms, they behave like other objects on the stack.

Since keyed cryptosystems contain excessive use of calculations (e.g., XOR operations) and inherently require that keys remain intact during use, some values on the cryptographic function's stack frame will inevitably fluctuate, whereas the key will likely *remain constant*. Thus, if we would pour a set of consecutive stackshots into a funnel that filters infrequent values from frequent, we would essentially delimit areas likely to contain the key. To achieve such a funnel, we use pattern mining. Specifically, since stack frames are sequentially allocated and compartmentalized, we use Sequential Pattern Mining (SPM) [71] to exploit the sequential ordering property.

In SPM, a subsequence is a *sequential pattern* (Definition 2.5.1) if it appears frequently in a dataset D, and its frequency is no less than a *minimum support threshold* (*minsup*), i.e., $\geq$ *minsup* of stackshots must overlap with the KEW for keys to become frequent (appear as a candidate). Although several SPM approaches exist, many have the critical drawback of presenting too many patterns [71]. We opted for Maximal Sequential Patterns (MSP, Definition 2.5.2), which have also been used to find the frequent longest common subsequences to sentences in texts and to analyze DNA sequences [71]. MSP mining is appropriate since it presents: (i) a concise subset of *unique* patterns [72], which prevents running a VSW over the same data unnecessarily and (ii) allows us to constrain the number of permitted *gaps* (irregular words) between consecutive words in a pattern. Since keys must usually remain intact, we use (ii) to require that each consecutive word in a pattern also appears consecutively in a stackshot (i.e., *no gaps*).

The conjunction of both properties (i-ii) enables exploitation of the stack's nature, where, within small time windows, some values remain constant (e.g., return addresses and *keys*) while others (e.g., temporaries in calculations) fluctuate. Note that since we use fluctuations to split the search space, the omission of values due to compiler optimizations (e.g., inlining and use of registers, see [167, 41]) can affect efficiency.

Definition 2.5.1 (Sequential patterns). A sequence dataset D is an unordered set of sequences: $D = \{S_1, S_2, \ldots, S_s\}$, where each sequence $S = \langle W_1, W_2, \ldots, W_n \rangle$ corresponds to one *stackshot* and consists of an ordered list of words W_i (2 bytes in MSP430), where i denotes its index. A sequence $S_a = \langle A_1\ , A_2, \ldots, A_m \rangle$ is *contained* in another sequence $S_b = \langle B_1, B_2, \ldots, B_n \rangle$ if there exists integers $1 \leq i < j < \ldots < k \leq n$ such that $A_1 = B_i, A_2 = B_j, \ldots, A_m = B_k$, and is denoted as $S_a \sqsubseteq S_b$. Here, S_b is a *super pattern* of S_a, while S_a is a *sub pattern* of S_b. A sequential pattern P is a sequence that is contained in one or more sequences in D.

Definition 2.5.2 (Maximal sequential patterns). A pattern P_a is said to be *closed* if there is no other pattern P_b, such that P_b is a *super pattern* of P_a, $P_a \sqsubseteq P_b$, and their support is equal. A pattern P_a is said to be *maximal* if there is no other pattern P_b, such that P_b is a *super pattern* of P_a, $P_a \sqsubseteq P_b$ [72].

2.6 An Architectural Blueprint

We proceed by consolidating the conceptualized methodology (Section 2.5) into concrete stages that can be translated into injectable malcode or stitched together from existing code, depending on which vulnerabilities $\mathcal{A}$ exploits (Section 2.4).

2.6.1 A High-Level Overview

Figure 2.5 depicts the consolidation of the conceptualized methodology (Section 2.5). The attack commences by $\mathcal{A}$ injecting (or stitching together) and triggering some malcode, which, once activated, embeds system hooks to proactively secure its *timely* invocation (Section 2.5.2). In the optimal case where it manages to attach callbacks directly onto the reception ISR (RX ISR), the malcode proceeds to remain stealthy until reception occurs. Upon reception, the reception interrupt flag (RXIFG, Step 1.1a) transitions, which causes the RX ISR to be invoked and call the malcode (Steps 1.2a and 1.3a). However, if hooking onto the RX ISR is too difficult, the malcode resorts to establish a periodic timer (T, Step 1.1b), where, on subsequent firings of the timer (Steps 1.2b through 1.4b), the malcode polls some pre-identified MMIO transceiver state (flag) (Section 2.5.2) to determine whether reception is occurring (Step 1.5b). Regardless of the method, once the reception has occurred, the malcode establishes a system timer to take stackshots periodically (Step 2). On each firing of the timer (Steps 3 through 5), the malcode extracts a predefined range *rg* from where SP currently points and stores it within some unused memory region (Step 6) - e.g., the *heap*, since OS-support for dynamic memory allocation in MCUS is often lacking. When a sufficient number of stackshots have been accumulated, they are marshaled into packets and transmitted back to $\mathcal{A}$ (Step 7), where $\mathcal{A}$ applies the search space reduction (Step 8). Finally, $\mathcal{A}$ concludes by hammering the remaining search space (e.g., by applying a VSW) for keys.

Note that $\mathcal{A}$ must dissect the target system's underpinnings to set up the necessary callbacks and timers. For the interested reader, see the technical details in Appendix A.

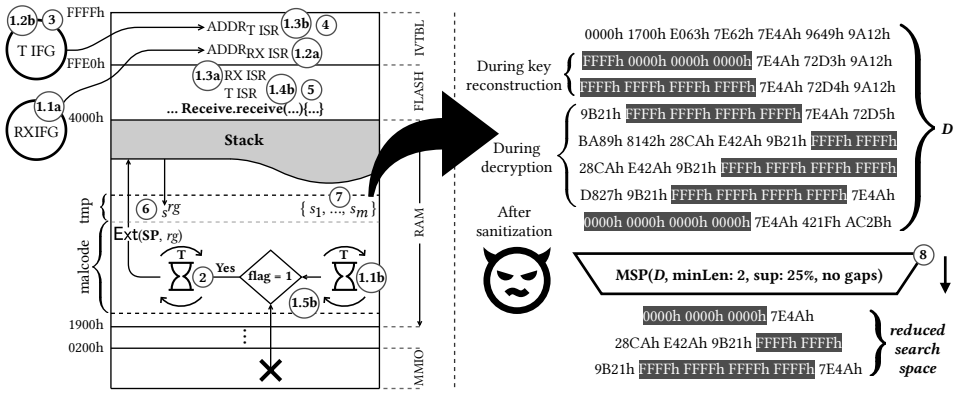


Figure 2.5: Holistic work-flow of the key-acquisition attack.

2.6.2 Modular Building Blocks

On a coarse-grained level, we separate the stages (Figure 2.5) into three versatile components. Two are interdependent and constitute the malcode – the *Watchdog* ($\mathcal{WD}$) and the *Frame Extractor* ($\mathcal{FE}$). The final component, the *Space Reductor* ($\mathcal{SR}$), is the utilization of MSP data mining (Section 2.5.5).

- **Watchdog ($\mathcal{WD}$).** The purpose of $\mathcal{WD}$ is to commence a system timer to periodically invoke $\mathcal{FE}$ when key exposure is imminent. Since we target the *software-oblivious* causality between reception and keyed cryptosystems, we can narrow the time window by periodically polling the transceiver's MMIO states (*observant* mode) or riding the RX-designated ISR (*dormant* mode). Although either approach effectively aligns the stackshots close to the KEW, dormant behavior is more stealthy since the CPU is free to enter (and stay uninterrupted) in low power modes (LPMs), but observer behavior might be easier to accomplish. For the target system, we opted for a *hybrid* construct to maximize timeliness, where $\mathcal{WD}$ rides the RX ISR and, upon invocation, starts a system timer to invoke $\mathcal{FE}$ periodically. However, instead of taking stackshots immediately, $\mathcal{FE}$ resists until a bit at a fixed memory location (which reflects whether transceiver communication is occurring) transitions, shifting stackshots closer to the reception handler's invocation (see Appendix A.1).
- **Frame Extractor ($\mathcal{FE}$).** The $\mathcal{FE}$ takes stackshots and transmits them to $\mathcal{A}$ once it has accumulated a certain amount. Each stackshot comprises the region between the current SP and SP $\pm$ *rg* bytes, depending on the direction of the stack. Note that although increasing *rg* improves the probability of obtaining $\mathcal{K}$ it also affects the amount of data to transmit, which translates to more memory resources and increases the possibility of detection.
- **Space Reductor ($\mathcal{SR}$).** Given stackshots D, MSP mining (Section 2.5.5) is used to get a reduced search space (Figure 2.5). Note that we demonstrate the practicality of $\mathcal{SR}$ in Section 2.7.

2.7 Experiments and Evaluation

We proceed to evaluate the $\mathcal{SR}$'s *efficiency* and *effectiveness*, where *efficiency* is quantified by the degree of the search space reduction and *effectiveness* by the ratio of keys in the initial (i.e., before applying $\mathcal{SR}$) and reduced search spaces.

2.7.1 Experimental Setup

We consider the reception handler in Fig. 2.4, where $\mathcal{F}$ is substituted with two prominent open-source implementations of the Advanced Encryption Standard (AES) algorithm, namely: TinyAES [111] and one developed by Texas Instruments [100] (hereinafter referred to as TIAES). In short, the AES algorithm operates on 128-bit blocks, accepts key sizes of either 128, 192, or 256 bits, and comprises several layers that are applied to manipulate an input block in several successive rounds N_r, where N_r is a function of the key length (10, 12, or 14 rounds for 128-, 192-, or 256-bit keys). However, because TIAES

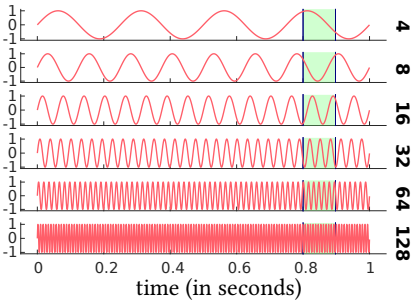
works exclusively with 128-bit keys, we restrict both implementations to use 128-bit keys (AES-128). Nonetheless, for each round, AES derives (expands) a separate round key (subkey) from the initial key (master key) using its *key scheduling algorithm* (resulting in a total of N_r+1 subkeys, where N_r=10 for AES-128), which it supplies to the *key addition layer*. However, whether the entire key schedule (set of subkeys) is precomputed or subkeys are derived as needed is a design choice. For example, since TIAES is developed to accommodate memory scarcity, it reuses the same 16-byte memory area of the master key for holding subsequent subkeys during run-time. Contrarily, TinyAES targets energy scarcity and therefore precomputes the key schedule, enabling reusing the same schedule on subsequent executions. However, since the schedule in TinyAES is stored consecutive to the master key, it only increases the master key's size, and given its dominating size, necessitates a large *rg* for us also to capture surrounding values. Thus, for practical reasons, the schedule is kept *above the function level* (Section 2.5.3), such that only the master key occurs on the stack. Therefore, for TinyAES, we assess $\mathcal{SR}$'s *effectiveness* by its ability to retain the master key in the reduced search space. For TIAES, however, we assess $\mathcal{SR}$'s *effectiveness* by its ability to retain *any* subkey (master key inclusive), which is justifiable since the remaining schedule can be inferred (though the amount of key bytes required to infer the remaining schedule differs between AES key sizes). Table 2.2 shows each AES implementation's cycle-accurate benchmarking results when executed within an emulated environment using MSPDebug v0.25 [17] in AES-128-ECB/decryption mode (including key expansion for TinyAES).

Table 2.2: Cycle Count (CC) and Execution Time (ET) in milliseconds for decrypting one 128-bit block at different optimization levels and considering a CPU running at 4 MHz.

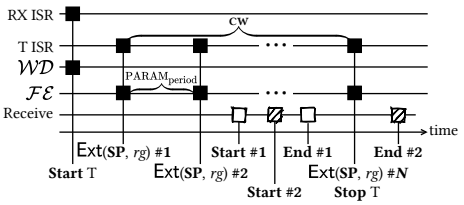
Implementation		*O0*	*O1*	*O2*	*O3*	*Os*
TinyAES	CC	390,279	34,408	28,986	23,612	33,234
	ET	97.57	8.602	7.247	5.903	8.309
TIAES	CC	52,256	16,642	13,411	8,505	15,747
	ET	13.064	4.16	3.353	2.126	3.937

To facilitate our experiments, we define a Capture Window (CW) as the *estimated* KEW and Capture Frequency (CF) as the number of stackshots per CW, which enables us to assess the stack at different temporal granularities by increasing and decreasing the CF. In general, CW must be large enough (over-approximated) or positioned close enough to overlap with $\mathcal{K}$'s KEW, and CF be high enough to ensure sufficient $\mathcal{K}$ captures ($\geq$ the *minsup*, Section 2.5.5) once the windows overlap (see Fig. 2.6a).

Note, however, that the subkey lifespan is drastically different for TinyAES and TIAES. For TinyAES, the master key $\mathcal{K}$ is kept intact, and its KEW will therefore be $\approx$ the emulated ET (Table 2.2). Thus, for a CW $\leq$ ET, aligned perfectly with the actual KEW, we could confidently use any relative *minsup* $\leq 100\%$ to reduce the search space without losing $\mathcal{K}$. However, for TIAES, each of the eleven subkeys will have KEW's of $\approx$ ET/11 (the KEW of some subkeys will, however, be different since not all rounds are identical). Hence, in the same setup, where CW $\leq$ ET is aligned with the beginning of decryption, then a relative *minsup* $\leq 9.09\%$ should suffice to retain all subkeys in the reduced search space so long as CF ≥ 11. Nonetheless, for experimental purposes, we let 25% and 6.25%



(a) How increasing the capture frequency (CF) enables us to capture better the key exposure windows (KEW, highlighted in green). Note here that the capture window (CW) is 1 second but the *actual* exposure window is ≈ 97.57 ms.



(b) Timeline (CF = ***N***) of how a CW might overlap the reception handler differently depending on when it begins.

Figure 2.6: Illustration of the capture frequency 2.6a and capture window 2.6b in a run.

(1/16) be our sufficiently permissive relative *minsup* thresholds for TinyAES and TIAES, i.e., $\mathcal{K}$ must appear in $\geq 25\%$ of stackshots for TinyAES and $\geq 6.25\%$ for TIAES.

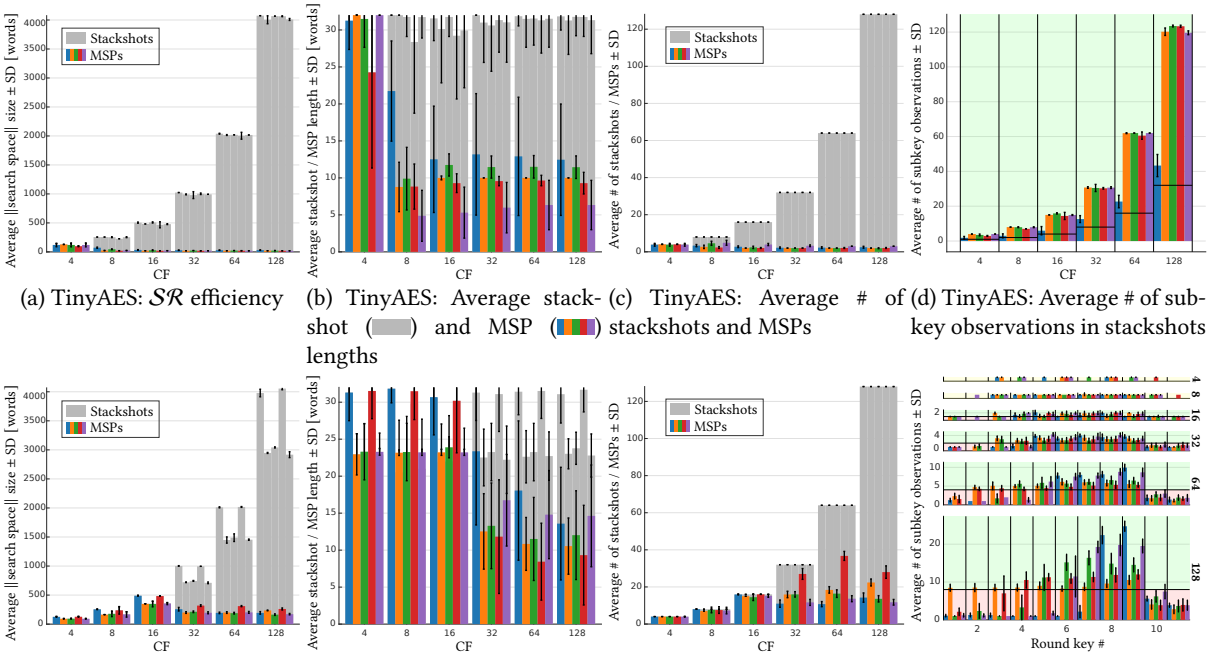
Furthermore, given the high ET variation between implementations and optimization levels (see Table 2.2), we consider a separate CW for each combination (i.e., CW = ET). Although these CWs can be considered optimal, note that ET ≠ KEW and ET neglects delays until the reception handler is invoked (might take several μs), the key reconstruction time, and interrupt processing time – which increases slightly as CF increases. Thus, stackshots will inevitably occur at different offsets from the reception handler (see Fig. 2.6b), and the CW will drift from the ET as CF increases. To reason about potential correlations between the CF and our experimental metrics – *efficiency* and *effectiveness* – we consider a set of CFs: $\{4, 8, 16, 32, 64, 128\}$. We use a common *rg* of 32 words (64 bytes) as the key is expected to occur relatively close to SP (see Section 2.5.3), and a minimum pattern length of 2 words.

2.7.1.1 Data Acquisition

We devised two scripts [55]: `getStacks`, for acquiring stackshots (using the malcode in Appendix A) from a Tmote Sky attached via an MSP430-JTAG (MSP-FET430UIF), and `spaceReductor` for MSP mining using seqwog v3.16 [26]. The `getStacks` script repeats n times (we set n = 15), where it: for each combination of AES implementation, CF and optimization level, (i) erases the memory of the Tmote Sky, (ii) compiles and programs the reception handler with TinyOS v2.1.2 [175] on the Tmote Sky, (iii) extracts $\text{ADDR}_{\text{restore}}$ (see Table A.1 in Appendix A) from the handler's assembly code, (iv) adjusts the malcode according to the CF and $\text{ADDR}_{\text{restore}}$, and writes it into memory—emulating *injection* (Section 2.4)—, (v) starts the Tmote Sky CPU, waits, and then stops the CPU, and (vi) finally reads the accumulated stackshots to a file, i.e., performs the data exfiltration (Fig. 2.5). Note, to trigger the reception handler, we setup another Tmote Sky to transmit packets periodically. Also, to emulate code injection and commence the mal-

code, we added an inline assembly instruction at the tail of the reception handler, which branches to $\text{ADDR}_{\mathcal{SE}}$ at the *first* invocation of the handler. Once `getStacks` completed, `spaceReductor` was supplied with the 900 *independent* datasets, on which it: (i) preprocessed each dataset by pruning repetitions of "FF3Fh" from stackshot tails, since these words are read *past* the stack boundary (38FFh for the MSP430-F1611 MCU), and (ii) applied the MSP mining and yielded the reduced datasets.

2.7.2 Empirical Results and Analysis



(a) TinyAES: $\mathcal{SR}$ efficiency (b) TinyAES: Average stackshot ( ) and MSP ( ) lengths (c) TinyAES: Average # of stackshots and MSPs (d) TinyAES: Average # of subkey observations in stackshots

(e) TIAES: $\mathcal{SR}$ efficiency (f) TIAES: Average stackshot ( ) and MSP ( ) lengths (g) TIAES: Average # of stackshots and MSPs (h) TIAES: Average # of subkey observations in stackshots

Figure 2.7: Efficiency and effectiveness of the MSP data mining ($\mathcal{SR}$). The figure shows means and Standard Deviations (SDs) of 15 independent datasets per combination of: AES implementation, CF and optimization level (*O0* ; *O1* ; *O2* ; *O3* ; *Os*).

We proceed by considering Fig. 2.7, which presents details about the extracted datasets, for each CF, before and after MSP mining with relative *minsup* thresholds of 25% and 6.25% for TinyAES and TIAES, respectively (implementations separated by rows and bars colored by optimization level).

2.7.2.1 *The case of TinyAES*

Plot 2.7a illustrates how efficiently the search space is reduced in the case of TinyAES (average reduction of 79.95%). Note that while $\mathcal{A}$ would run a VSW separately on each stackshot or MSP to find keys, Plot 2.7a considers the *concatenated* (||) search space, i.e.,

the concatenation of all stackshots in a dataset before and all MSPs after applying $\mathcal{SR}$. Nonetheless, it is clear that even as we increase the CF significantly, the *reduced* search space, comprising the MSPs found in a dataset (batch of stackshots), remains considerably small (average size of 40.8 words). Furthermore, note that in the particular case when CF is 4, the reduction is expected to be negligible for a relative *minsup* of 25%, since the search space is only reduced when some patterns: (i) are infrequent, i.e., occur in $< 25\%$ stackshots, which is not possible with four stackshots, or (ii) appear in multiple stackshots – as only the *super pattern* is presented (Section 2.5.5). Excluding the insignificant CF of 4, the efficiency increases to 95.27%, and the average *reduced* search space decreases to 25.64 words. To better illustrate the effect of the data mining on a stackshot/MSP level, Plots 2.7b and 2.7c give further insights on the average stackshot/MSP lengths and count, respectively.

Regarding effectiveness (accuracy), Plot 2.7d illustrates how often the *entirety* of the master key appears on average in the datasets before the data mining, and the black horizontal lines denote the considered relative *minsup* threshold of 25%. We can see that the key almost always appears as *frequent* (above the threshold) and that for most optimization levels, we could confidently raise the *minsup* threshold without losing the key. Note that the key is observed less often for optimization level `O0` because its final word lies on the edge of our *rg* of 32 words, i.e., many stackshots only include the partial key.

2.7.2.2 The case of TIAES

As with TinyAES, Plot 2.7e illustrates the reduction of the *concatenated* search space when using the relative *minsup* of 6.25% (average reduction of 42.87% and an average search space size of 228.31 words), and Plots 2.7f and 2.7g show the average stackshot/MSP lengths and count. As before, we expect negligible search space reduction for CFs $\{4, 8, 16\}$, since the support threshold comes into effect when the number of stackshots transcends 16. Excluding these CFs, we gain efficiency of 84.17% and a reduced search space size of 220.30 words. Note that although this search space is indeed much larger than for TinyAES, it contains *several* subkeys.

Regarding effectiveness, Plot 2.7h details how often the *entirety* of subkeys (the first subkey is the master key) appear on average in the datasets before data mining, and the black horizontal lines denote the considered relative *minsup* threshold of 6.25%. Note that the latter half of the subkeys appear more frequently than others because when AES runs in decryption mode, it applies subkeys in reverse order (the master key *last*). Scrutinizing the results reveals that we seem to lose some subkeys at different optimization levels as we increase the CF. This trend occurs because our static CWs overlap less with the ET as we increase the CF due to the system spending more time processing interrupts (Section 2.7.1) – including the execution of some parts of the malcode. Nonetheless, despite TIAES's design approach making the key extraction more complicated – since the KEW of subkeys is *much* smaller than that of TinyAES's master key (Section 2.7.1) –, the fact that some subkey always occurs above the horizontal line (the *minsup* threshold), indicates that *at least* one subkey always appears in the reduced search space, as part of an MSP.

2.8 Discussion and Potential Defences

We have demonstrated how an $\mathcal{A}$ can systematically acquire *highly* ephemeral keys in MCUS. To make matters worse, for asymmetric cryptosystems, the Key Exposure Problem (KEP) is even more extended, e.g., considering the F1611 CPU @ 8 MHz, 1024-bit RSA encryption takes hundreds of milliseconds and decryption several seconds [86]. Since we target keys *during* use, sanitizing keys *after* use is *insufficient*, and so is keeping keys *above the function level* as the key's address on the stack is exploitable and challenging to avert (see Section 2.5.3). However, since the attack's effectiveness relies on our ability to approximate the use of keyed cryptography post-reception and incorporate it as a trigger mechanism, any distortion of the approximation (e.g., delayed processing) mitigates the attack. Further, since carrying out the attack is difficult without disturbing program execution, Control-Flow Integrity (CFI) solutions can be utilized to prohibit the attack, and Control-Flow Attestation [2] can be used to *detect* whether an attack has occurred. Nonetheless, such defensive and solutions are *attack-specific* and do not directly resolve the KEP. To resolve the KEP, we must ensure that keys are either *useless when observed (captured)* or *unobservable*.

2.8.1 Resolving the Key Exposure Problem

To resolve the KEP, we could decide never to store keys sequentially in memory. By (i) storing key bytes in different endianness, (ii) permuting the bytes' order, or (iii) scattering bytes throughout memory, we can directly affect $\mathcal{A}$'s ability to use the key. However, although (i) and (ii) prevent a naive VSW attack, they fail to prevent $\mathcal{A}$ from plainly trying all byte (or word) permutations. The latter method (iii) ultimately *compels* $\mathcal{A}$ to resort to an exhaustive attack since the search space might never contain all of the required pieces. However, realizing either method requires careful code instrumentation. Comparable strategies include white-boxing [48] and Moving Target Defense (MTD) [165]. With white-boxing, a given key is transformed into code that performs cryptographic operations without using the key material explicitly. However, embedment of keys into software makes key revocation difficult and is generally unsuitable in environments where keys must be frequently updated. With MTD, we make the key a moving target by rearranging its bytes regularly during run-time, which can be a viable mitigation tactic – assuming that the rearrangement scatters the bytes and not only permutes their order. Toward unobservability using software, we could decide to confine cryptographic keys to CPU registers. For example, TRESOR [136] proposed storing the key inside debug registers, Loop-Amnesia [166] inside MSRs, AESSE [135] inside SSE registers, and [146] inside SSE XMM. However, achieving *secure* CPU-bound keys is presumptuous since it requires a sufficient number of *special* registers that are guaranteed to remain inaccessible to the adversary.

2.8.2 Prominent Hardware-Entangled Guards

Mechanisms based purely on software are desirable but have proven unsatisfactory. Furthermore, in TinyOS, the advised approach is for the CC2420 radio chip to hold the cryptographic keys and then perform in-line encryption and decryption of packets. However, although this approach seems to make the keys *unobservable*, they can unfortunately be

effortlessly requested back using SPI [101] due to the lack of protection on the target device. Therefore, it is better to utilize more comprehensive solutions, e.g., Trusted Execution Environments (TEEs, e.g., ARM's TrustZone) and hardwareized cryptographic modules (e.g., Trusted Platform Modules), which provide hardened interfaces for secure storage and use. Though rarely available in resource-constrained commodity devices or MCUs due to their weight and cost, such trusted computing solutions can perform all necessary cryptographic operations while ensuring that keys *remain unobservable*. Another, arguably more lightweight and cheap alternative, would be to utilize Physically Unclonable Functions (PUFs) [129] as a form of secure key storage. Although PUFs inherently result in some form of key exposure window as the cryptographic keys must be reconstructed before use, it has been shown that PUFs can be entangled with existing cryptographic primitives [129], leading to so-called hardware-entangled cryptography, which eliminates the need to store keys in memory altogether. Further, as demonstrated by [108], PUFs can also be integrated into the processor's instruction pipeline to allow run-time execution of encrypted code, making it difficult (if not futile) for an adversary to attempt to observe softwareized keys on a target system.

2.9 Conclusions

There is a lack of adequate containment and trust regarding an embedded system's behavior, where sophisticated software attacks can circumvent standard key management techniques. By exploiting the strong causality between reception and keyed cryptosystems and the fact that cryptosystem implementations inherently require keys to remain exposed in memory during use, we have demonstrated how keys can be timely acquired off the memory stack during run-time. Our work serves a three-fold purpose: to reveal how the determinism of wireless-capable (event-driven) MCUS can pose an exploitable threat, study the effects of severe key-exposure attacks, and motivate the need for lightweight KEP-resilient protocols in resource-constrained MCUS.

Chapter 3

ZEKRO: Zero-Knowledge Proof of Integrity Conformance

Abstract

In the race toward next-generation systems of systems, the adoption of edge and cloud computing is escalating to deliver the underpinning end-to-end services. To safeguard the increasing attack landscape, remote attestation lets a verifier reason about the state of an untrusted remote prover. However, for most schemes, verifiability is only established under the *omniscient and trusted verifier* assumption, where a verifier knows the prover's trusted states, and the prover must reveal evidence about its current state. This assumption severely challenges upscaling, inherently limits eligible verifiers, and naturally prohibits adoption in public-facing security-critical networks. To meet current zero trust paradigms, we propose a general ZEro-Knowledge pRoof of cOnformance (ZEKRO) scheme, which considers mutually distrusting participants and enables a prover to convince an *untrusted* verifier about its state's correctness in zero-knowledge, i.e., without revealing anything about its state.

3.1 Introduction

To make cloud computing services more resilient to increasing integrity concerns and enable detection of malicious or vulnerable software (e.g., Apache Log4j [143]), several proposals [7, 184, 156, 126, 127] have advocated leveraging trusted computing technology. This technology relies on a Trusted Execution Environment (TEE) or more commonly a Trusted Platform Module (TPM) [172] deployed on each node which acts as a trust anchor to store and report integrity evidence about the node's configuration. When a node's software stack boots up, it chronologically measures (hashes) each software component and extends each measurement into a Platform Configuration Register (PCR) of its TPM. To report the node's configuration integrity, the TPM uses a unique key to sign the aggregated PCR value, which a remote verifier can then compare against a list of trusted reference values to determine whether it corresponds to a trusted state. For en-

hanced integrity guarantees, we can also have nodes continue measuring their software beyond the boot process, e.g., using the Integrity Measurement Architecture (IMA) [155] or Policy-reduced IMA (PRIMA) [103].

Despite their benefits, most existing remote attestation protocols suffer scalability issues since the complexity of the verifier grows with the complexity of the prover's configuration [128]. Specifically, the verifier must continuously maintain an extensive allowlist of trusted reference values for each prover, which becomes prohibitively impractical for large networks. Besides inherent scalability issues, it is also easy to prove that protocols that require the prover to disclose its configuration state to verifiers fail under the honest-but-curious adversarial model [148] since curious verifiers can easily link and identify the software executing on a prover. Such failure to respect a prover's configuration privacy is known to raise serious privacy issues, such as discrimination [153] and can even foster dedicated software attacks against a vulnerable prover [185]. While catering to a platform's configuration privacy is essential from a security perspective, omitting the exchange of such information can also promote services to mix more seamlessly in multi-domain coordinated services, including multi-vendor environments [15], where contractual differences could otherwise prohibit such collaboration.

To remediate the privacy issue and simultaneously reduce the verifier complexity in remote attestation, some proposals [185, 45, 9, 7, 184, 126, 127] offer different, more privacy-respecting mechanisms for a verifier to reason about a prover's correctness. However, these schemes tend to either require an intermediate trusted third party (TTP) between the prover and verifier to distill or hide the integrity report from the verifier [184, 7], which limits network design and overall responsiveness, assume that the prover's configuration is translatable into abstract properties which the verifier is knowledgeable enough to interpret [45, 9], restrict who can verify a particular prover [185, 127], incur high overhead [121], or consider only the load time measurements of a prover's software stack [126].

3.1.1 Contributions

This paper presents ZEKRO, a zero-knowledge proof of conformance scheme that uses trusted computing abstractions to overcome the barriers of configuration privacy and scalability. These abstractions provide another building block for constructing scalable services that seamlessly mix in multi-domain environments and are more resilient to integrity concerns. Our design includes two crucial main innovations to overcome the limitations of existing TPM-based privacy-respecting remote attestation protocols.

First, the ZEKRO scheme provides the trusted computing abstraction, called *policy-restricted attestation key*, that dynamically restricts a node's attestation key (secured in its TPM) to policies chosen by an authorizing entity (e.g., a domain orchestrator). By predicating its ability to use its attestation key for signing on its configuration correctness, we can verify a node's conformance using a simple challenge-response protocol that neither requires nor reveals any configuration information. Moreover, to update a node's "trusted configuration state" during runtime (e.g., in response to patches), the authorizing entity can reactively or proactively authorize new policies restricting the node's attestation key to a new configuration state. Second, to control which of the already authorized policies a node can satisfy during attestation, we propose creating policies that additionally require explicit, time-limited authorization, called *leases*, to be satisfiable, which allows an

authorizing entity to control which policy can temporarily be satisfied.

To demonstrate ZEKRO's performance, we evaluated a proof-of-concept prototype, whose implementation is made publicly available [57] to ensure reproducibility and verifiability of our results.

3.2 Related Works

When measuring a node's software during runtime, one idea, which IMA [155] employs, is to record measurements both into a TPM PCR and a Measurement Log (ML). Due to the unpredictable order in which components are loaded, the ML helps verifiers verify the reported aggregated PCR value and whether each loaded component is trusted based on reference values. However, such information disclosure raises concern, especially if measurements belong to different tenants sharing a platform. To make it more privacy-respecting, Container-IMA [127] proposed generating unique secrets to obscure ML entries related to each tenant's software, thus effectively restricting verification of a particular tenant's software to verifiers that know the secret, in addition to the trusted reference values.

Similarly, to restrict verifiability only to "correct" verifiers, authors of [185] propose obfuscating a PCR by recording random values into the PCR and ML, which ensures that only a legitimate verifier can perform the verification and others learn nothing. However, the problems of dishonest verifiers and verifier complexity remain.

To avoid disclosing configuration measurements (digests) and reduce the verifier complexity, Property-Based Attestation (PBA) [153, 45] maps configurations to more semantical requirements, called "properties", which a prover can prove to fulfill without disclosing its concrete configuration. However, bridging the semantic gap between digests and properties, i.e., identifying what maps to which properties, is nontrivial and must be agreed upon beforehand. The same applies to similar techniques, such as [9], where the authors propose grouping sets of software versions using cryptographic functions such as chameleon hashes [116], or group signatures [44].

To unload verifiers, authors of [7] propose having an intermediate third party between a prover and verifier who is trusted to verify the prover's measurements and vouch for the prover's integrity. Similarly, the work in [184] proposes an attestation proxy to mediate attestation requests between the prover and verifier and translate the prover's concrete configuration into properties that are returned to the verifier. However, while effective, such approaches limit network flexibility in practice and incur overhead.

To reduce the involvement of the trusted third party, the work in [126] proposed having the party only initially involved in instructing a node in sealing (i.e., encrypting) a secret signing key to its trusted configuration state using its TPM. This way, a node can only ever unseal (decrypt) its secret key if its configuration has not changed. Thus, verifiers, who know a node's public key, can determine its integrity simply by requesting it to sign a challenge using its secret key. However, there are some inherent security issues with the sealing operation. First, the unsealed key is exposed to software during attestation. Second, as pointed out in [153], any configuration update (e.g., patches) would render it impossible to unseal the key.

Therefore, to prevent exposing the signing key to software, the authors of [121] proposed instead to create attestation keys, which is a special type of asymmetric keys cre-

ated inside a TPM, where the secret part that is used for signing never leaves the protective shielding of the TPM. Then, similar to [126], these attestation keys are created under the supervision of a trusted third party with an authorization policy that constrains the use of the secret part to a particular configuration state. However, due to the brittleness of the authorization policy, nodes require new attestation keys whenever their configuration changes, resulting in high computational and network overhead. Furthermore, the approach disregards adversaries blocking update requests to lock a node in a "trusted state" and lacks authentication to withstand adversaries impersonating the entity responsible for measuring a node's configuration, effectively allowing adversaries to hide by reporting false measurements.

To solve these problems, we propose (i) creating attestation keys with "flexible" authorization policies that allow a trusted authority (e.g., a domain orchestrator) to approve new policies that restrict the same key's use to new configuration states, (ii) enforcing a leasing mechanism on the approved policies to ensure freshness, and (iii) authenticating the entity responsible for reporting measurements.

3.3 Trusted Computing Concepts

This section presents the background necessary to understand the proposed approach by describing the leveraged functionalities.

3.3.1 Enhanced Authorization

The TPM can be used as a combination lock for securing access to TPM objects, such as cryptographic keys. To specify under which circumstances an object can be accessed (i.e., what should be fulfilled before access is granted) and what operations are permitted once access is granted, we must first create a policy statement that logically describes all of our conditions and the scope of the authorization. To ensure statement interpretability, we must use the available Enhanced Authorization (EA) TPM commands, which are documented in part 3 of TCG's specification [172]. For example, to allow the use of an object only if a PCR has a specific value, we can use the *PolicyPCR* command to reference which PCR should have what value. We then translate our policy statement into an "policy digest", which is computed by aggregating a digest over each EA command's Command Code (CC) and command-specific arguments (like an onion) as detailed in the documentation [172]. With a policy digest, we can ask the TPM to create an object (e.g., an attestation key) with this policy digest as its authorization policy.

To satisfy our object's authorization policy, we must start a "policy session" with the TPM, to which it will associate a policy digest internally. The rules are simple. To satisfy our object's authorization policy, we must invoke the correct combination of EA commands and arguments to make the session's internal policy digest match our object's authorization policy. Anytime we invoke the TPM to perform some action using our object, e.g., to sign a message, the TPM will first check that the current session's internal policy digest satisfies the object's authorization policy. For EA commands that restrict an object's scope to specific commands and arguments, another session-specific digest called the command parameter hash $cpHash$ is also used and checked before performing an action.

3.3.1.1 Flexible Policy

This paper aims to use the EA functionality to create an attestation key whose authorization policy is constrained to a node's correct configuration as measured into some PCR. Unfortunately, this is not possible with the *PolicyPCR* command since it only allows one state. While we could create a policy statement to permit several possible states by logically *or*ing several *PolicyPCR* commands, this requires knowing all future "good" states, making it inherently impractical. Fortunately, there is a workaround to this "brittleness" problem using the *PolicyAuthorize* command, which allows creating a "flexible" policy owned by a secret key whose public key is associated with the policy. In other words, whoever owns the corresponding secret key of the public key that we commit to in the policy digest has complete control to sign (approve) policies during runtime that, when satisfied in a TPM session, will also cause the object's authorization policy to be satisfied. Suppose we only have the *PolicyAuthorize* command as part of our policy statement. In that case, the resulting "flexible" policy digest pol is computed as $pol \leftarrow \mathsf{H}(\mathsf{H}(CC_{PolicyAuthorize} \parallel \mathsf{pk}) \parallel ref)$, where pk is the public key, and ref is an optional reference that restricts the authorization policy. For example, we use each node's unique identifier as the corresponding reference value to distinguish between the authorization policy of different nodes in a domain.

Once we have a flexible policy, we can have the node create an attestation key with a flexible authorization policy, thus allowing the respective orchestrator to approve different restrictions to use the key, e.g., depending on its currently accepted configuration.

To approve a policy, the orchestrator first creates the policy digest to approve $aPol$ and then signs an authorization hash $aHash \leftarrow \mathsf{H}(aPol \parallel ref)$ using its secret key, where ref references the node.

Finally, to use the approved policy to satisfy the attestation key's authorization policy, the node first satisfies $aPol$ in a policy session and then calls *PolicyAuthorize* with $aPol$, a public key pk, its identifier ref, and a proof that $\mathsf{H}(aPol \parallel ref)$ was signed by the owner of pk, which proves whether the current session's policy digest was approved. Then, if it holds, the TPM replaces the session's policy digest with $\mathsf{H}(\mathsf{H}(CC_{PolicyAuthorize} \parallel \mathsf{pk}) \parallel ref)$, which unlocks the use of the attestation key if it matches its authorization policy.

3.3.2 Trustworthy Runtime Measurements

While the TPM provides secure storage and reporting, the entity that stores measurements into the TPM should be trustworthy. For example, to create a chain of trust of the integrity of a platform's boot sequence, we could have each component, such as firmware and boot drivers, first measure the next component into a TPM PCR before passing control to that component. However, if we cannot trust the code that measures the first component, called the core root of trust for measurements, we cannot trust the PCR aggregate.

The most prominent method of extending such a chain of trust into the operating system is IMA [155]. When IMA is used, it hooks onto file-related system calls to remeasure a file (part of the trusted computing base) into an ML and a PCR whenever the file is accessed. To only remeasure a file when it is modified, the filesystem must support i_version and, if needed (e.g., EXT3, EXT4), be mounted with this option. When enforced, the filesystem updates the i_version field of the inode associated with a file when a file

is modified.

As before, if we cannot trust IMA, we cannot trust its measurements. Similarly, we must also rely on a trusted entity to measure a node's configuration. This trusted entity must be isolated and immutable. Note, however, that the choice of isolation, e.g., OS-based process isolation, user/kernel-level isolation, or hypervisor-based approaches, will depend on use-case-specific requirements. We assume some Trusted Execution Environment (TEE) for this paper, like ARM TrustZone or Intel SGX. However, note that since we consider remote TEE invocation, it raises two problems. First, requests can be blocked by an adversary to evade detection. Second, the adversary can spoof measurements. To prevent both attacks, we must ensure that the measurements are authentic and approved policies are only temporarily satisfiable. One way to achieve the first requirement, which we utilize, is to create a TPM object inside the TPM's non-volatile (NV) memory space of type PCR, which allows us to associate an authorization policy to our PCR object as described in Section 3.3.1 that allows the node's TEE to authenticate the measurements. Our solution to the second requirement is to enforce a *leasing* mechanism, which we describe in Section 3.5.4.2.

3.3.3 Zero-Touch Enrollment

An attestation key (AK) is especially beneficial due to its inherent restriction. Whereas unrestricted keys, when created inside a TPM, can be used to sign any data, an AK, which is restricted, will not sign any externally provided data structure that appears to be valid and TPM produced but is not. Thus, if an AK is known to be protected by a TPM, it may be relied on to report that TPM's contents accurately.

In the context of zero-touch provisioning of a remote platform equipped with a TPM, we must first verify the authenticity of the TPM [174, 173, 181] and all other primitives that enable the subsequent attestation. The core of zero-touch provisioning is the correct creation of the attestation key to secure the integrity of the attestation process. For discrete TPMs, the manufacturer generally installs an Endorsement Key (EK) and an associated certificate inside the TPM, which allows verifying the EK's authenticity and can further be used to create a Local Attestation Key (LAK). However, the EK differs from an AK. An EK is a "storage key" used to protect (encrypt) the secret key of other keys to allow safe storage outside the TPM, and creating a LAK based on the EK requires some extra steps [174].

Another method is to install an Initial Attestation Key (IAK) and associated certificate, which can be used to create the LAK. Specifically, by utilizing the inherent characteristics of attestation keys, we can, after verifying the IAK's certificate [174], verify that a LAK is created in the same TPM by certifying it using the IAK.

3.4 System and Threat Model

Before we delve into details of the ZEKRO scheme, we present the considered setting and assumptions regarding protocol participants.

3.4.1 System Model and Security Assumptions

We consider a network setting with three types of entities:

1. **Prover** is an untrusted node equipped with a secure TPM provisioned with a certified Initial Attestation Key (IAK) and a secure element for providing secure runtime measurements. We consider a TEE with a certified key pair for brevity for the rest of the paper. Finally, to detect file modifications, we assume a trusted filesystem that enforces i_version.

2. **Verifier** is an untrusted node that knows the orchestrator of a prover node and wants to remotely check the correctness of the prover's configuration (though not limited to this role).

3. **Orchestrator** is a trusted entity that: (i) onboards each node by verifying its initial key certificates, (ii) performs the zero-touch configuration of each node's LAK, and (iii) maintains and approves each node's acceptable configuration. We assume that the orchestrator has already done the onboarding (i), which lets us instead focus on more relevant challenges.

3.4.2 Threat Model

We consider a software adversary who, on some prover, exploits a software vulnerability that allows it to modify that node's critical configurations that are part of the Trusted Computing Base (TCB) as determined by the domain orchestrator. The adversary's goal is to remain undetected and may even attempt to disrupt the node's communication with the orchestrator to do so. Further, we assume that verifiers are dishonest and attempt to infer information about the prover's configuration. However, we assume that verifiers do not collude with the prover's adversary to obtain prover information.

3.4.3 Protocol Objectives

Our scheme's objectives are threefold: (i) any tampering of a prover node's TCB configurations or continued disobedience is detectable, (ii) verifiers require no knowledge except the trusted key certificate of a prover node's orchestrator to verify the prover, and (iii) verifiers learn no information from the verification process besides the correctness of the prover's configuration integrity. Note that while we consider challenges of continuous integrity verification, we defer the problem of continuous, in-memory runtime attestation. **This is covered by the next paper described in Chapter 4.**

3.5 The Protocol

We continue with the terminology used in our description of the ZEKRO scheme. Then we proceed to give an overview of the scheme before diving into its protocols with explicit reference to the required TPM commands found in part 3 of the TCG specification [172].

3.5.1 Notation

As an accompanying reference while reading the protocol diagrams, we consider the following symbols and simplified TPM terminology:

$\mathsf{H}(m)$	Compute m's digest using collision-resistant hash function H.
$\mathsf{Sign}(k, m)$	Compute a cryptographic signature over m using k.
$\mathfrak{T}(@conf)$	Tracer, which, given a path, returns a tuple $(c, ino, iver)$ with the contents c, inode number ino, and inode version $iver$.
$\mathsf{Vf}(expr)$	Verification of $expr$, which interrupts if the evaluation fails.
PCR	Platform Configuration Register, which is an extend-only structure internal to the TPM: $PCR \leftarrow \mathsf{H}(PCR \parallel someVal)$
$mPCR$	Mock PCR, which mimics a PCR and is used by the orchestrator and provers to maintain the expected value of the NV PCR that is used for recording measurements (needed for attestation).
CID	Configuration identifier associated with a $mPCR$ on the orchestrator to reference a prover's current configuration version.
ID	A unique (prover) node identifier.
$\mathfrak{H}$	Handle, which references an internally loaded TPM object.
$tmpl$	Template for a TPM object that describes its type and attributes.
CC_{cmd}	Command Code of the TPM command cmd.
exp	Expiration time in some unit of time.
ref	Policy reference used in policies to differentiate authorizations.
pol	Policy digest as described in Section 3.3.1, which, for our purposes, is a chain of computations: $pol \leftarrow \mathsf{H}(\mathsf{H}(pol \parallel CC_{cmd} \parallel name) \parallel ref)$, where $name$ denotes a TPM object's (e.g., a key) name, which is generally a digest of its public area.
$cpHash$	Command parameter hash, which is computed from the parameters of a TPM command as described in part 1 of TCG's specification [172] and, for our purposes, is computed as: $cpHash \leftarrow \mathsf{H}(CC_{cmd} \parallel params)$, where $params$ refers to the command-specific parameters. With a $cpHash$, an authorizing entity can restrict policies to a specific command and arguments.
$aHash$	Authorization hash as described in part 3 of TCG's specification [172], which for the: $PolicyAuthorize$ command has the form $aHash \leftarrow \mathsf{H}(aPol \parallel ref)$ to enable an authorizing entity to dynamically approve a policy $aPol$ to use the object, and for the $PolicySigned$ command has the form $aHash \leftarrow \mathsf{H}(n \parallel exp \parallel cpHash \parallel ref)$ to enable signed authorization for $expirable$ execution of $cpHash$ in a TPM session whose nonce is n.
tkt	Ticket, which the TPM computes for a specific command and its arguments that later proves to the same TPM that it has already performed the necessary (signature) verification.
$\{\mathsf{opk}, \mathsf{osk}\}$	The orchestrator's asymmetric keypair.
$\{\mathsf{tpk}, \mathsf{tsk}\}$	A TEE's certified asymmetric keypair.
AK	Attestation Key, a restricted signing key that can sign internal TPM structures and has a public part $AKpub$ and private part $AKpriv$, where $AKpriv$ never leaves a TPM unencrypted.
IAK	Initial AK, which is initially created and certified inside a TPM.

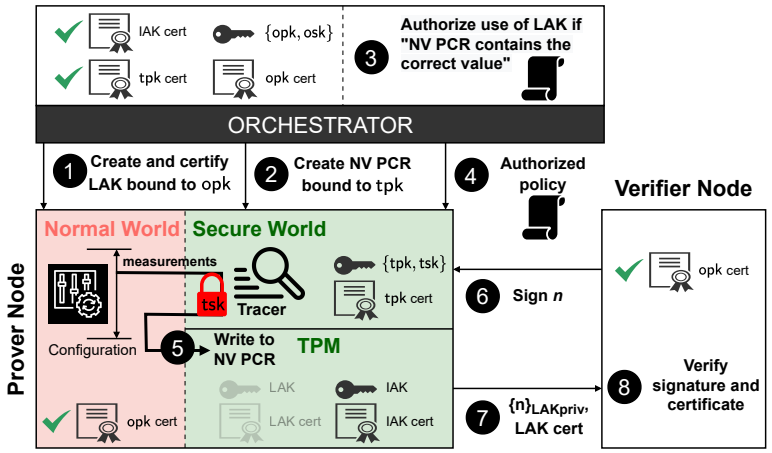


Figure 3.1: System model and conceptual work-flow after the orchestrator has verified a prover's TPM and TEE keys.

LAK Local AK, which is remotely certified to reside inside a node's TPM by the local (domain) orchestrator.

SK Storage Key, which is a restricted decryption key used as a parent to wrap (encrypt) the private part of descendant keys, thus ensuring child key secrecy when stored outside the TPM.

3.5.2 High-Level Overview

Fig. 3.1 shows the conceptual work-flow of the ZEKRO scheme considering the different entities described in Section 3.4.1. Our scheme aims to facilitate zero-touch enrollment and configuration of deployed nodes that execute privacy-critical services such that anyone can efficiently verify the correctness of the service in zero-knowledge, thus enabling stitching of privacy-respecting services. Note that, for generality, our scheme succeeds the initial onboarding of the nodes, where a domain orchestrator verifies a node's TPM and TEE certificates and (possibly) verifies that its filesystem enforces i_version. The scheme comprises three protocols: (i) prover enrollment, (ii) configuration update, and (iii) oblivious remote attestation. Let us start by clarifying the idea of each protocol while regarding Fig. 3.1.

3.5.2.1 Enrollment

To enable a deployed node to become a "publicly" verifiable prover, it must first be securely enrolled by a certified orchestrator that will then be responsible for continuously determining the node's correct state, whose truthfulness is asserted to verifiers in zero-knowledge. For this, the orchestrator configures the node's TPM with two objects: (i) a local attestation key (LAK) created with a flexible authorization policy with the orchestrator as its authorizing entity (step 1), and (ii) a non-volatile (NV)-based PCR created with an authorization policy that binds the node's TEE as its authorizing entity (step 2). The former assures that only the orchestrator may approve policies that permit the use

of the LAK. Similarly, the latter ensures that only the TEE may approve policies that permit modification of the NV PCR. Together, the TPM objects enable the orchestrator to continuously and securely predicate the use of the LAK to the node's currently trusted configuration state by approving policies that require the NV PCR to contain the currently expected (trusted) aggregate value, which is securely maintained by the node's TEE. Finally, although omitted from Fig. 3.1, the node must initially report the unique number and current version of the inodes assigned to its configuration files to allow the orchestrator to include inode information in approved policies to ensure detection of unauthorized file modifications. Note that if the initially reported information is incorrect, the prover cannot satisfy approved policies since the TEE's measurements would cause the NV PCR to have a different value since the TEE always uses inode information currently associated with the files.

3.5.2.2 Configuration Update

Once a prover is successfully enrolled, its orchestrator's responsibility is to approve policies that restrict the prover's use of its LAK, which can occur either routinely or upon demand, e.g., due to a newly released patch. To approve policies for a specific prover node, the orchestrator keeps a trusted reference copy of that node's currently correct configuration, including the associated inode information, and a mock PCR, which it uses to deterministically compute the *expected* value of the TEE's NV PCR once it has remeasured the node's configuration (step 3).

Upon receiving a new approved policy for its LAK (step 4), the prover invokes a trusted application running inside the Secure World of its TEE, which: (i) securely measures the requested configuration, (ii) authorizes a one-use policy for extending the measurement into its NV PCR, and (iii) returns the measurement and corresponding authorized policy to the prover's Normal World, where the prover uses the authorization to extend the measurement into the NV PCR on behalf of the TEE (step 5). Note that we elaborate on the motivation behind outsourcing the extension of the NV PCR to the prover's Normal World in Section 3.5.3.2. Furthermore, to protect against an adversary blocking the orchestrator's request to measure the node's configuration from reaching the node's TEE in an attempt to evade detection, we describe in Section 3.5.4.4 the utilization of an accompanying leasing mechanism that enables the orchestrator to grant the node only temporary ability to satisfy a selected approved policy.

3.5.2.3 Attestation

The final, oblivious remote attestation protocol is executed solely between a prover and a verifier, where the verifier represents anyone that, trusting the orchestrator, wishes to determine the correctness of the prover. Like any remote attestation protocol, the verifier initiates the execution of the protocol by challenging the prover with a fresh nonce (step 6). Then, due to the nature of the LAK object and its strong dependency on the TEE's NV PCR, the verifier knows that if the prover can correctly present a signature over the nonce using a LAK that was certified by the orchestrator (steps 7 and 8), then this serves as irrefutable evidence that it satisfies whichever policy that the orchestrator approved. Thus, without knowing any of the prover's configuration details or what is executing on the prover, and without requiring the prover to disclose any information, the verifier is

convinced, in zero-knowledge, that the prover's configuration is correct. Conversely, if the prover cannot supply such a signature, then the verifier can reasonably assume that the prover cannot satisfy the orchestrator's policy. Note, however, that the freshness of the prover's assertion is directly correlated to the orchestrator's frequency of approving new policies, i.e., a higher update frequency leads to faster detection.

3.5.3 Prover Enrollment

While the creation of a LAK and an NV PCR are both subsumed under the initial enrollment protocol as described in Section 3.5.2.1, we here separate them for clarity since the creation of each object requires different operations that are unique to that specific object.

3.5.3.1 Secure Local Attestation Key Creation

Fig. 3.2 shows how the orchestrator verifies that a node has correctly created a LAK in the same TPM as the pre-provisioned IAK that was verified during the node's initial onboarding, as follows. First, to prepare an authorization policy for the LAK which ensures that the orchestrator is the object's only authorizing entity, the orchestrator composes a flexible policy digest which: (i) binds the orchestrator as the object's authorizing entity, and (ii) includes a reference to the specific node's unique identifier (*ID*) which allows the orchestrator, who potentially orchestrates several nodes, to distinguish between authorizations. Then, to inform the node's TPM about how it should create the LAK, including the LAK's attributes and authorization policy, the orchestrator sends a generic LAK template together with the prepared authorization policy to the prover, who passes these values as arguments in a call to its TPM to create the LAK. However, because the TPM's storage capacity is severely scarce, it cannot be used to store several keys persistently. Thus, to protect the LAK's private part when it is stored external to the TPM, it is created as a child of some storage key *SK*, where the purpose of *SK* is to wrap (encrypt) the LAK's private part for safe storage outside the TPM.

To prove that the LAK was created correctly, the prover loads it back into its TPM, where the same *SK* is used internally to decrypt its private part, and a handle referencing the loaded key is returned that enables cryptographic operations targeting the LAK. Then, to prove that the LAK resides in the same TPM as the IAK, the IAK is used to sign a TPM-generated certificate that includes all creation details of the LAK. The certificate and signature are then sent back to the orchestrator for validation and assurance that the key has the correct characteristics. Finally, if everything holds, the orchestrator signs a certificate for the LAK using its secret key, which it gives to the prover node to show to verifiers to prove its LAK's validity.

3.5.3.2 Secure NV PCR Creation

Like the creation of the LAK, Fig. 3.3 shows the creation of the NV PCR, where, instead of the orchestrator, the node's TEE is now appointed as the object's authorizing entity to ensure authentic measurements. The orchestrator begins by preparing an object template that instructs the node's TPM when defining the NV object's characteristics, e.g.,

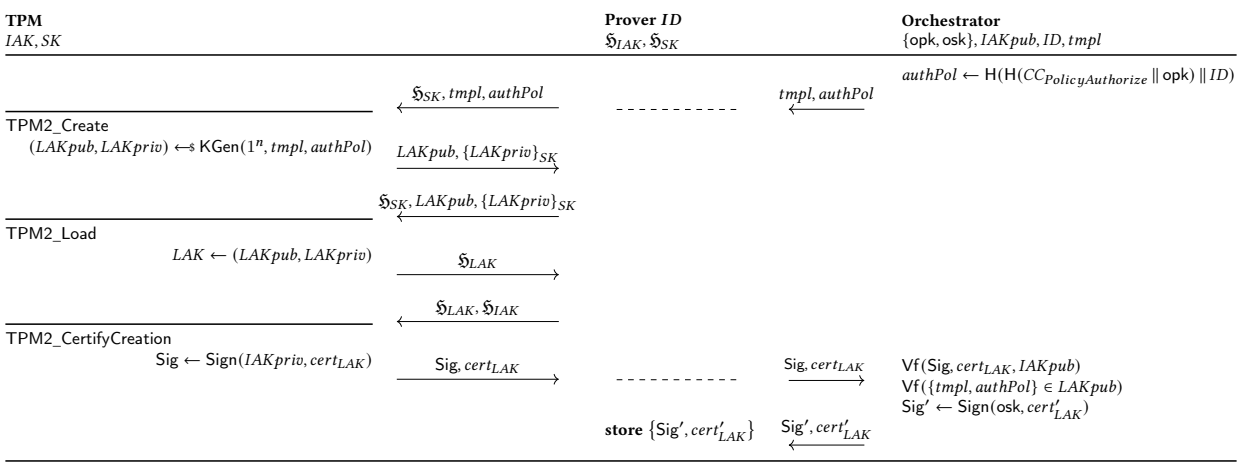


Figure 3.2: Creation of a LAK with a flexible authorization policy based on an IAK.

that it should behave like a PCR. This object template, together with an authorization policy requiring signed authorization from the node's TEE to modify (extend or delete) the object, an index to reference the created NV object, and an initial value *iv* that should be initially extended into the NV PCR, are then sent to the prover node. Given these values, the prover first calls its TPM to create the NV PCR. Then, to extend the initial value into the created NV PCR, which requires authorization from the TEE, the prover starts a TPM session and then passes that session's nonce together with the initial value and NV PCR index to a trusted application executing inside the Secure World of its TEE. To authorize the prover to extend the NV PCR only once and only with the correct value, the trusted application composes a command parameter hash *cpHash* to restrict its authorization to a single TPM command and arguments, namely the *NV_extend* command with the initial value and NV PCR index as arguments. Then, to provide signed authorization for the *cpHash*, the trusted application signs an authorization hash *aHash* over the *cpHash* and the prover's session nonce using its secret key that restricts the authorization only to the currently active session.

Note that if the trusted application could communicate directly with the TPM (or we had chosen another isolation mechanism), we would not necessarily need the command parameter hash. However, it is impractical to communicate with the TPM directly from within a trusted application due to their inherently small codebase and limited APIs, and it would require a large chunk of the TPM software stack (TSS) to manage an entire TPM session. Therefore, we instead have the TEE authorize permission to extend the measurement into the NV PCR, which can be outsourced to the Normal World since it is nonreusable and cannot be used to extend incorrect values.

Given signed authorization from the TEE, the prover runs the *PolicySigned* command with the TEE's signature, its active session nonce, *cpHash*, and a handle to the TEE's public key. Assuming the authorization was correctly signed, the *PolicySigned* command updates the session's policy digest and sets the session's command parameter hash to the authorized *cpHash*, thus restricting which command the prover can execute. Then, to extend the TEE's NV PCR, the prover runs *NV_Extend*, where, assuming the session's

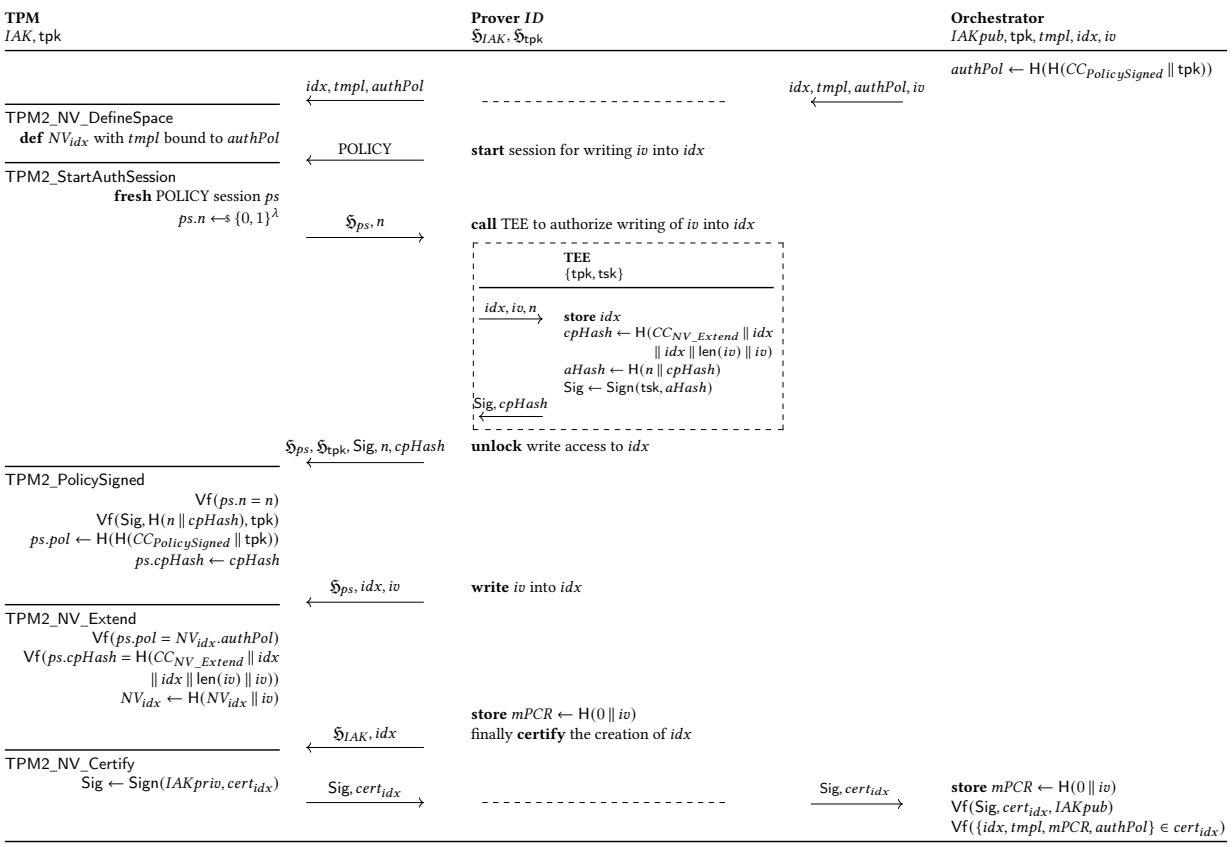


Figure 3.3: Creation of a NV PCR controlled by the prover TEE's public key.

policy digest matches the NV PCR's authorization policy and the command arguments match the session's command parameter hash, the TPM will update the NV PCR with the provided initial value. Afterward, to keep track of the value of the NV PCR, the prover records the extension in its local mock PCR. Finally, similar to the LAK creation process described in Section 3.5.3.1, the prover proves that the NV PCR was created correctly by certifying it using its IAK, which the orchestrator verifies by inspecting the signed certificate.

3.5.4 Configuration Update

Securely equipped with a LAK and an NV PCR, Fig. 3.4 shows how the orchestrator can reliably approve policies that permit the prover to use its LAK only if its current configuration is correct by predicating policies on the authentic contents of the TEE's NV PCR, as follows.

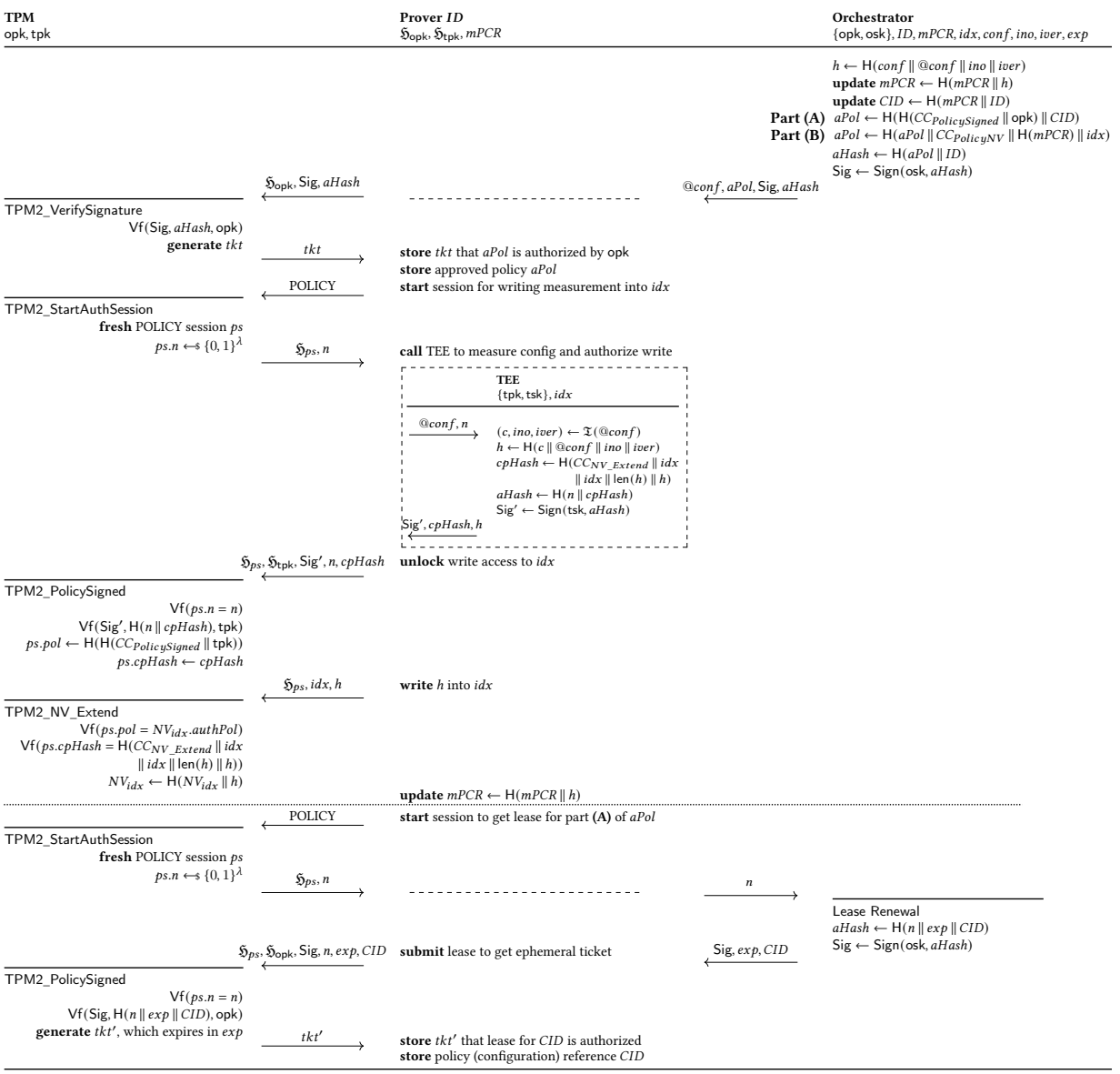


Figure 3.4: To enforce a new configuration state on a prover node, the orchestrator approves a policy for the prover node's LAK that is (i) restricted to the new configuration state and (ii) requires time-limited authorization to use.

3.5.4.1 *Calculating the Golden Hash*

Let us assume that the orchestrator wants to allow a prover to use its LAK if its current configuration is correct. To simplify the discussion, we narrow the scope of the considered configuration to one file whose filesystem location on the prover is $@conf$. To create

such a policy, the orchestrator first computes the expected (trusted) value of the prover TEE's NV PCR after it has been extended with the correct measurement. To compute this value, the orchestrator computes a hash over the contents of the currently correct version of the prover's configuration file, denoted $conf$, including its known location $@conf$, inode number ino, and current inode version $iver$, which it extends into its local mock PCR. Note that by including (committing to) the file location, we effectively prevent any adversary from feeding the prover's TEE with a spoofed path since the TEE's measurement would differ. Similarly, the inclusion of the inode number and version ensure that any unexpected deletion or modification of the configuration file is detected. Finally, the orchestrator computes a unique configuration identifier CID as a hash over the current value of the mock PCR and the node's identifier (ID), which now uniquely references the prover's current configuration version.

3.5.4.2 Approving the Policy

To approve a policy for the node's LAK that is unique to the current configuration identifier CID and requires that the TEE's NV PCR contains the currently expected value, the orchestrator creates an approval policy $aPol$ requiring that the node presents: (a) signed authorization from the orchestrator with explicit mention of CID and (b) that the TEE's NV PCR contains the expected value. To authorize the approved policy, the orchestrator signs an authorization hash $aHash$ over the approved policy and a reference to the specific node's ID, such that the approved policy will work (match) only with that node's LAK. Note here that the purpose of the first part (a) of the approved policy is to allow the orchestrator to enforce a leasing mechanism on its approved policies. Specifically, for a node to satisfy an approved policy $aPol$ whose part (a) references some CID, the orchestrator must sign an authorization hash referencing CID. Furthermore, by including an expiration in the authorization hash, a node's TPM will revoke the authorization after the specified time, thus requiring a node to request a new "lease" to continue satisfying an approved policy. Finally, because the orchestrator controls which CID it authorizes during lease renewal and presumably chooses the most recent, we castrate any adversary attempting to lock a node in a "good" state.

3.5.4.3 Configuration Remeasurement

Once approved, the orchestrator sends the policy, its signed authorization hash, and the considered configuration file(s) path to the appropriate node. To ensure the signature's authenticity, the node uses its TPM to verify it under the orchestrator's public key. However, since verification is expensive, and the node must prove the approved policy's authenticity every time it attempts to satisfy it to use its LAK, the TPM outputs a ticket if the verification was successful, which is then later supplied as evidence to the TPM that it has already done the verification.

Then, to remeasure its configuration (or part thereof), the node starts a TPM session and passes the session's nonce with the configuration file path in a call to the trusted application executing inside the Secure World of its TEE. Then, using its accurate tracer mechanism, the trusted application: (i) retrieves the specified file's content and inode information and then (ii) computes a hash (measurement) over the contents, its filesystem location, and the associated inode number and version. Finally, similarly to how it

authorized extending an initial value into its NV PCR as described in Section 3.5.3.2, the trusted application signs an authorization hash for a one-time policy to extend the computed measurement, which the prover node then uses to update the NV PCR and also its mock PCR.

3.5.4.4 Leasing

Finally, depending on the update frequency and the considered expiration time for leases, the prover must repeatedly request new leases from the orchestrator to continue satisfying part (a) of the currently approved policy. To get a new lease, the prover must start a new TPM session and send the session's nonce to the orchestrator, who then signs an authorization hash that includes the session nonce, some expiration time, and the current configuration identifier *CID*. These values, together with the signed authorization hash, are then returned to the prover who, in the same TPM session, must pass them in a call to *PolicySigned*, where, if everything holds, a self-expiring ticket will be returned that proves to the TPM that the orchestrator signed these specific arguments, which can be used to temporarily satisfy part (a) of the currently approved policy.

3.5.5 Oblivious Remote Attestation

When challenging a prover with a nonce, Fig. 3.5 shows how the prover attempts to satisfy an approved policy *aPol* to use its LAK to sign the nonce. To satisfy *aPol*, the prover starts a new policy session, over which it runs (i) *PolicyTicket* with a ticket proving that it has signed authorization from the orchestrator, and (ii) *PolicyNV* with the index of its TEE's NV PCR and the expected value, which it maintains in its mock PCR. If the ticket is correct and has not expired, the TPM updates the session's policy digest with a value corresponding to part (a) of *aPol*. Similarly, if the expected value matches the actual PCR contents, the TPM updates the session's policy digest with a value corresponding to part (b) of *aPol*. Finally, assuming that the session's aggregated policy digest matches *aPol*, the prover can run *PolicyAuthorize* with its *ID*, *aPol*, the orchestrator's public key, and a ticket proving *aPol* was signed by the orchestrator. If everything holds, the TPM replaces the session's policy digest with the "flexible policy digest", which specifies the orchestrator's public key as the authorizing authority and references the node's *ID*. Assuming the session's policy digest matches the LAK's authorization policy, the prover can sign the nonce using its LAK and subsequently send the signature and its LAK certificate back to the verifier in a single pass, where, if everything holds, the verifier is convinced that the prover is currently in a correct state.

3.6 Empirical Performance Evaluation

3.6.1 Implementation and Experimental Setup

We implemented our protocols described in Section 3.5 in C++ with IBM's TPM Software Stack (TSS) v1.6.0 [80] and OpenSSL v1.1.1i, compiled using the GNU GCC compiler. We considered only elliptic curve keys and used SHA256 exclusively as our hashing function. We evaluated our dockerized implementation [57] on a Raspberry Pi 4 Model B platform

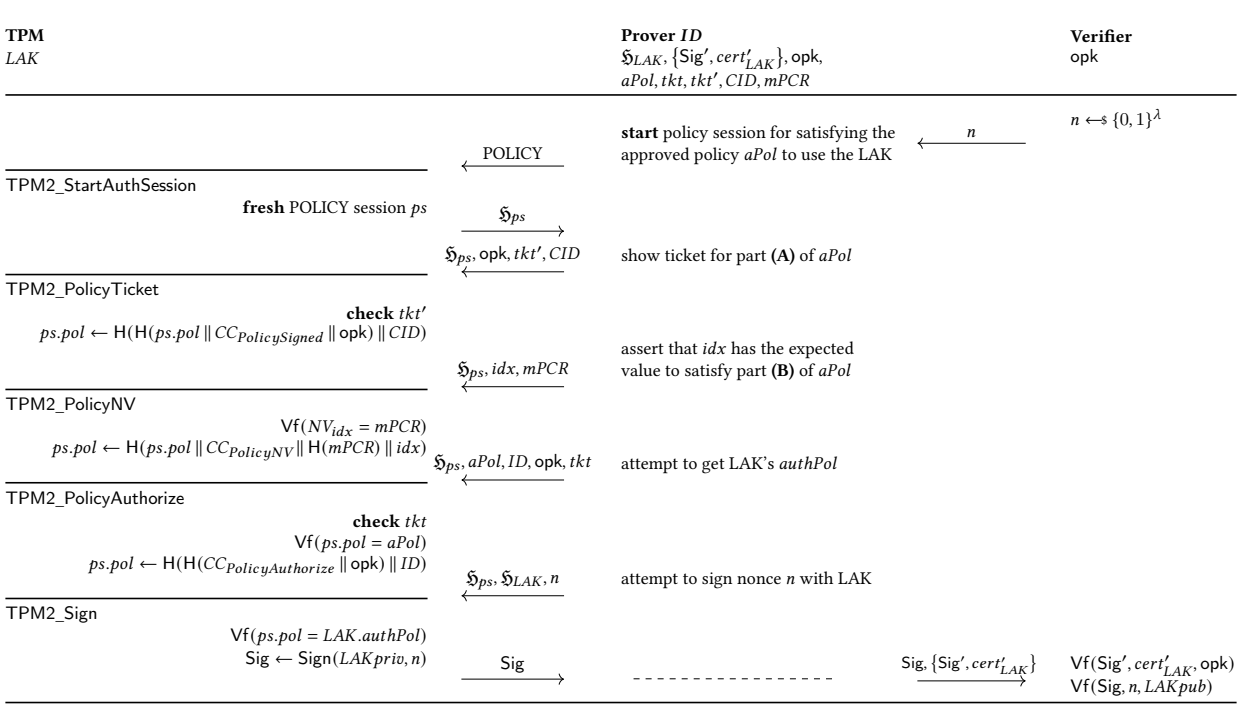


Figure 3.5: The Oblivious Remote Attestation protocol, where a verifier makes initial contact to a prover and challenges it to prove its configuration integrity by signing a nonce using a LAK certified by the prover's orchestrator.

with a 1.5 GHz ARM processor, where we recorded the protocol's performance against IBM's software TPM v1637 [80] and an OPTIGA SLB9670 hardware TPM.

3.6.2 Performance Benchmarks

Our performance results are summarized in Table 3.1, where, for each subprotocol, we show: how long it takes to complete the protocol (first row) and how much time it takes for the completion of each of the TPM commands when executed against either a hardware or software TPM (next rows). Note that the timings are produced using C++11's chrono library's system clock, and each timing statistic also includes the time spent by the TSS to perform the necessary processing of our commands and its internal session management. Furthermore, in the case of the hardware TPM, each timing statistic also includes any Low Pin Count (LPC) bus delay.

Note that we omitted the timings for the orchestrator's verification of the created LAK and NV PCR and the verifier's verification of the signed nonce since these operations do not necessarily require interaction with the TPM and are near-instant operations. It is also worth mentioning that, despite the empirical findings showing that a hardware TPM is an efficiency barrier, it provides security guarantees against stronger adversaries than a software TPM.

Regarding our protocols, note that the first two protocols, i.e., the protocols for creat-

Table 3.1: Performance of the protocols over 15 iterations. The table shows the average time in milliseconds (and standard deviation, σ) to run each of the protocols at a prover when considering a software (left) and hardware TPM (right).

Subprotocol	Avg. (SWTPM)	Avg. (HWTPM)
LAK creation	14.98 ($\sigma \approx 0.96$)	699.16 ($\sigma \approx 10.90$)
TPM2_Create	9.68 ($\sigma \approx 1.61$)	267.11 ($\sigma \approx 2.34$)
TPM2_Load	1.07 ($\sigma \approx 0.25$)	154.21 ($\sigma \approx 26.73$)
TPM2_CertifyCreation	4.22 ($\sigma \approx 1.02$)	277.83 ($\sigma \approx 3.62$)
Attaching a NV PCR	10.67 ($\sigma \approx 0.48$)	576.43 ($\sigma \approx 6.35$)
TPM2_NV_DefineSpace	1.24 ($\sigma \approx 0.23$)	26.70 ($\sigma \approx 0.26$)
TPM2_StartAuthSession	0.96 ($\sigma \approx 0.24$)	31.60 ($\sigma \approx 0.91$)
TPM2_PolicySigned	3.26 ($\sigma \approx 0.79$)	272.51 ($\sigma \approx 26.13$)
TPM2_NV_Extend	1.31 ($\sigma \approx 0.37$)	44.20 ($\sigma \approx 2.30$)
TPM2_NV_Certify	3.91 ($\sigma \approx 0.75$)	201.42 ($\sigma \approx 2.16$)
Measurement update	12.98 ($\sigma \approx 0.61$)	832.41 ($\sigma \approx 7.70$)
TPM2_VerifySignature	3.24 ($\sigma \approx 1.02$)	179.99 ($\sigma \approx 1.47$)
TPM2_StartAuthSession (X2)	1.91 ($\sigma \approx 0.24$)	63.19 ($\sigma \approx 0.91$)
TPM2_PolicySigned (X2)	6.51 ($\sigma \approx 0.79$)	545.03 ($\sigma \approx 26.13$)
TPM2_NV_Extend	1.31 ($\sigma \approx 0.37$)	44.20 ($\sigma \approx 2.30$)
ORA	7.37 ($\sigma \approx 0.38$)	474.67 ($\sigma \approx 5.14$)
TPM2_StartAuthSession	0.96 ($\sigma \approx 0.24$)	31.60 ($\sigma \approx 0.91$)
TPM2_PolicyTicket	0.75 ($\sigma \approx 0.20$)	68.77 ($\sigma \approx 0.72$)
TPM2_PolicyNV	0.73 ($\sigma \approx 0.20$)	61.07 ($\sigma \approx 0.25$)
TPM2_PolicyAuthorize	0.72 ($\sigma \approx 0.19$)	68.74 ($\sigma \approx 0.58$)
TPM2_Sign	4.21 ($\sigma \approx 1.08$)	244.49 ($\sigma \approx 23.25$)

ing a LAK and the protocol for creating an NV PCR, need only be executed once for each node due to the flexibility of their authorization policies; thus, their performance is negligible. After that, the most time-consuming protocol is the configuration update protocol executed between a domain orchestrator and a node. However, note that the reported timings include the time for (i) verifying the orchestrator's newly approved policy, (ii) extending the node's measurements into the TEE's NV PCR, and (iii) using a lease from the orchestrator to get a ticket to satisfy the first part of the newly approved policy as shown in Fig. 3.4. If we only consider the time to get a new ticket for a new lease, which only requires one *StartAuthSession* command and one *PolicySigned* command, then the time is only $\approx$ 4.22 ms using the software TPM and $\approx$ 304.11 ms using the hardware TPM. Finally, the ORA protocol, which nodes execute among themselves, takes a prover ≈ 0.5 seconds to complete on a hardware TPM and < 10 ms with a software TPM.

3.7 Security Properties

Our scheme is designed to achieve the following security properties:

Property 3.7.1 (Secure Enrollment). During a node's zero-touch enrollment, its trusted orchestrator (i) verifies that it created a LAK inside the same authentic TPM as its certified IAK and (ii) certifies that the LAK only ever obeys policies approved by the node's trusted

orchestrator.

Property 3.7.2 (Policy Authenticity). By initially embedding a node's identifier in its LAK's authorization policy, we ensure that policies approved for a node's LAK by its orchestrator must bear that node's identifier to be satisfiable, thus effectively preventing any adversary from using policies meant for one node to unlock the LAK on a different node.

Property 3.7.3 (Measurement Authenticity). By predicating all policies approved for a node's LAK on an NV PCR that requires explicit TEE authorization for being written into and having the TEE only authorize its measurements, we prevent adversaries from supplying unauthentic measurements to unlock a LAK. Further, by including (committing to) filesystem path(s) and inode details of the considered configurations in the approved policies, we prevent path spoofing and ensure the detection of any unauthorized configuration modification that may have occurred since the last measurement, e.g., as a result of transient malware. Specifically, since the attacks would cause the TEE's measurements to differ from the expected measurements used by the orchestrator in the approved policy, an affected node would become unable to satisfy the approved policy to use its LAK to convince verifiers about its configuration correctness.

Property 3.7.4 (Policy Freshness). By predicating all policies approved for a node's LAK on additional authorization by the orchestrator with explicit mention of that node and the approved policy it applies to (through the configuration identifier) and having the orchestrator include expiration on such authorizations, we secure a leasing mechanism to prevent nodes from satisfying approved policies beyond their intended timeframes, thus preventing adversaries from satisfying policies approved for old configuration states in an attempt to evade the remeasurement process of the node's current configuration.

Property 3.7.5 (Implicit Revocation). Since continued use of a LAK requires continued orchestrator authorization, nodes halt unless kept alive.

Property 3.7.6 (Zero-Knowledge Verification). Most importantly, our oblivious attestation protocol ensures that verifiers need no reference materials to determine a prover node's correctness, nor can they infer anything about its configuration since the decision is solely based on the prover presenting a correct signature over a fresh challenge.

3.8 Conclusions

We presented ZEKRO, a novel, scalable, efficient, and effective orchestration scheme that utilizes state-of-the-art trusted computing technologies to facilitate the secure orchestration of a multitude of nodes over multi-domain networks while allowing mutually distrusting nodes to partake in privacy-preserving remote attestation activities to determine the configuration correctness of one another.

Chapter 4

ZEKRA: Zero-Knowledge Control-Flow Attestation

Abstract

To detect runtime attacks against programs running on a remote computing platform, Control-Flow Attestation (CFA) lets a (trusted) verifier determine the legality of the program's execution path, as recorded and reported by the remote platform (prover). However, besides complicating scalability due to verifier complexity, this assumption regarding the verifier's trustworthiness renders existing CFA schemes prone to privacy breaches and implementation disclosure attacks under "honest-but-curious" adversaries. Thus, to suppress sensitive details from the verifier, we propose to have the prover outsource the verification of the attested execution path to an intermediate worker of which the verifier only learns the result. However, since a worker might be dishonest about the outcome of the verification, we propose a purely cryptographical solution of transforming the verification of the attested execution path into a *verifiable computational task* that can be reliably outsourced to a worker without assuming any trusted execution environment on the worker. Specifically, we propose a novel method of encasing a program-agnostic execution path verification task inside an arithmetic circuit whose correct execution can be verified by untrusted verifiers using a zero-knowledge proof system without the verifiers learning any secret inputs. The asymptotic complexity of our construction, in terms of the outsourceable circuit's size, is $O(3N + E + 3E\sqrt{N} + E\sqrt{D})$, where E is the execution path size, N is the number of nodes in the attested program's Control-Flow Graph, and D is the shadow stack depth. We also benchmarked our construction under the Groth16 zkSNARK proof system, showing <10 seconds performance with an AMD Ryzen 7 3700X CPU to generate proofs with parameters E=1K,N=1K,D=15 that are verified in 2 ms.

4.1 Introduction

To safeguard the increasing computing system attack landscape [120, 112], traditional remote attestation schemes let a (trusted) verifier reason about the state of a remote

prover's computing platform. The security of such schemes generally relies on a trust anchor on the prover, capable of securely recording and authenticating platform evidence. Building on this concept, Control-Flow Attestation (CFA) [2, 63, 186, 62, 3, 176, 95, 125, 169, 117, 140, 141] aims to determine whether a program was executed correctly on a resource-constrained prover by verifying that no runtime attacks (e.g., ROP [162]) subverted the program's control-flow behavior. To ensure that a program was executed correctly, existing CFA schemes assume a trusted verifier who maintains complete reference materials, such as the program's Control-Flow Graph (CFG) and in-memory program layout, and other acceptance criteria to decide on the legality of the attested program's execution path, as recorded and reported by the prover's trust anchor. However, besides severely degrading scalability due to the incurred verifier complexity, the unattractive need to exchange comprehensive prover information discourages the adoption of CFA in both public-facing and emerging multi-domain services [15] where privacy constraints or contractual differences among vendors might prohibit such disclosure.

While not yet demonstrated for CFA, the concept of Property-Based Attestation (PBA) [45] could reduce the verifier complexity and prevent information disclosure by giving the verifier only the verification result in the form of some semantical property. However, performing the necessary verification and property translation locally on the prover would require a resourceful trust anchor capable of correctly maintaining all trusted reference materials *and performing the verification correctly*, which is sometimes impractical, especially for resource-constrained settings such as those generally considered in CFA. Here, the provers are severely underpowered devices equipped with carefully designed minimalistic trust anchors [63] whose sole purpose is to record and authenticate a program's execution path during attestation. Therefore, without complicating the prover, another option is to introduce an intermediate, more powerful party, which we refer to as a worker (sometimes called an "attestation proxy"), responsible for performing the attestation verification on behalf of the verifier and conveying only the result back to the verifier. However, to convince the verifier that the verification was done correctly, we would again generally encounter heavy assumptions, such as requiring trusted hardware or a Trusted Execution Environment (TEE) with attestable execution (e.g., Intel SGX) [177] to protect the verification process on the worker.

Instead, we propose to utilize Verifiable Computation (VC) to transform the task of verifying the prover's attested program execution into an outsourceable arithmetic circuit whose *proof of correct execution* can be generated by the worker and efficiently verified by the verifier, proving the attestation verification's correctness and outcome. Using VC, we *need no assumption of any trusted computing base on the worker* while also protecting against a wider range of attackers than approaches that consider some form of TEE. Further, to hide certain inputs of the proof generation (e.g., the attested execution path and program details) from *completely untrusted* verifiers, we use a privacy-enhanced VC scheme called zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARK) [83].

Motivating use case. One sample scenario is an IoT-based Air Quality Decision Support System (AQDSS) [151], wherein citizens employ smart sensors (publishers) that report their readings onto a blockchain, which can then be consumed by the public (subscribers). Here, the publisher devices are produced by trusted manufacturers, but we have no trust assumptions about the subscribers. Thus, while the public (subscribers) wants guarantees that the readings originate from uncompromised sensors, the manufacturers (or citizens)

might not want to disclose sensor details to the subscribers, hence the need for privacy. While [151] considered the case of providing evidence that the static state of a sensor device was uncompromised along with its readings, our approach gives additional evidence for the untrusted public to confirm that the expected function of the sensor was executed correctly (i.e., in the absence of any control-flow attack) without disclosing any sensitive information.

4.1.1 Contributions

We propose a novel protocol called *ZEro-Knowledge contRol-flow Attestation* (ZEKRA), which is, to our knowledge, the first privacy-preserving CFA protocol. Without imposing additional prover assumptions, we remove all trust and complexity assumptions regarding verifiers by outsourcing attestation verifications to intermediate workers who employ VC to convince verifiers about the verification results. Our work offers the following contributions: (i) We present a novel scheme that lets underpowered provers convince untrusted verifiers about a program's correct execution in zero-knowledge by offloading the verification to an intermediate worker that assures verifiers about the result without disclosing any secrets using zkSNARK technology; (ii) We detail our outsourceable circuit design, including the use of several circuit optimization techniques; (iii) Realistic case studies, showing how ZEKRA can resolve privacy issues in privacy-sensitive (*non*-time-critical) application domains; and, (iv) We validate and benchmark ZEKRA with a proof-of-concept implementation, which we make publicly available [56] to ensure reproducibility and encourage further work.

4.2 Related Works

Many prevention methods have been proposed to address program runtime attacks, e.g., shadow stacks and stack canaries [54], Control-Pointer Integrity (CPI) [118], and Control-Flow Integrity (CFI) [1, 183, 75, 85, 124, 107] all aim to protect a program's attack surface during runtime. However, these methods fail to provide any assurance to a remote entity since all enforcement happens locally and will (depending on the enforced policy) abruptly stop–and possibly crash–a device upon violations, which can be dangerous in certain safety-critical applications.

Runtime attestation. To *detect* control-flow attacks against a program, C-FLAT [2] proposed instrumenting the program to self-report its control-flow events during runtime to a Trusted Execution Environment (TEE), which is then hashed and reported to a trusted verifier who checks that the digest exists in a set of trusted reference values. LO-FAT [63] leverages a customized hardware module to intercept the executed instructions at runtime to improve C-FLAT performance. ATRIUM [186] enhances both C-FLAT and LO-FAT to detect TOCTOU attacks that swap malign program segments with benign segments during attestation to evade detection. It relies on a customized hardware module that runs attestation parallel to the main processor. ScaRR [176] aims to apply CFA to complex systems, e.g., cloud-native virtual machines. To deal with the challenge of representing all the valid execution paths in complex systems, ScaRR follows C-FLAT's approach of splitting the control-flow execution into sub-paths, where the idea is basically to record each unique loop path only once in the execution path while maintain-

ing associated counters to track the number of times each path was taken. Tiny-CFA [140] is a CFA protocol that relies on the APEX Proof-of-Execution (PoX) architecture [139] and, similar to C-FLAT, assumes that the software is instrumented. ReCFA [187] performs control-flow attestation of complex software by relying on the static binary analysis and binary instrumentation. In particular, the scheme compresses the control-flow evidence efficiently and enforces control-flow integrity policy at the binary level with a remote shadow stack. Note that other CFA schemes also exist, which additionally consider: utilizing machine learning [95], a log-based approach and use of physically unclonable functions [125], distributed settings and use of multiset hash function to reduce the size of the reported execution path [3], which despite its benefits, makes the verification unable to detect certain attacks since the order of the control-flow transfers is not preserved in the multiset representation. Some approaches also consider detecting some data-oriented attacks [62, 169, 117, 141] by verifying the integrity of both control-flow and data involved in the execution.

Note that our work is *complimentary to the above approaches*: prior work can leverage our scheme to weaken the trust assumptions concerning the verifier. Specifically, whereas prior CFA works generally focus on recording and reporting the program's execution path, we focus on the layer between provers and verifiers to remove the *omniscient and trusted verifier* assumption by making the verification zero-knowledge.

Verifiable computation. Unlike CFA schemes that consider underpowered provers and powerful verifiers, proof-based Verfiable Computation (VC) enables weak verifiers (our provers) to outsource computationally intensive computations to powerful yet untrusted provers (our workers) who return proof that the computation was done correctly. Moreover, to enable secret inputs in the computations, privacy-enhanced VC schemes, e.g., zkSNARKs [83], guarantee that the proof reveals nothing about the secret inputs. Furthermore, whereas proof generation is slow, verification is remarkably fast, making zkSNARKs attractive, especially in Distributed Ledger Technology (DLT), e.g., the anonymous cryptocurrency Zcash [157] and Ethereum.

To verify general programs using zkSNARKs, specialized compilers, such as TinyRAM [19], vnTinyRAM [20], and Buffet [180] have enabled the transformation of traditional programs into low-level circuits, whose execution can be proven and verified securely. For example, the TinyRAM [19] circuit compiler takes a high-level C program and a time-bound T as input and compiles the program to special assembly instructions, whose emulated execution on some input for up to T cycles in a general-purpose MIPS-like CPU, called TinyRAM, is expressed as an arithmetic circuit that verifies the correct execution of the input program. However, the principal disadvantage is cost since the number of circuit constraints (i.e., the size of the circuit) grows unwieldy as the program complexity increases, which strongly correlates to the amount of time it takes to generate proof over the circuit's execution. Improving on the per-cycle cost, TinyRAM's successor, vnTinyRAM [20], achieved a quasi-constant per-cycle cost of $\approx 1,458$ constraints, and later Buffet [180] further improved control flow and random memory access by 1-3 and 1-2 orders of magnitude, respectively.

Nonetheless, directly expressing general programs as circuits remains expensive. Fortunately, CFA schemes have demonstrated that verifying a program's execution is enough to convince a remote verifier that a program executed correctly with respect to its control flow. While CFA's security guarantee is only a subset of that of VC, this paper demonstrates that combining the two allows underpowered provers to prove a program's ex-

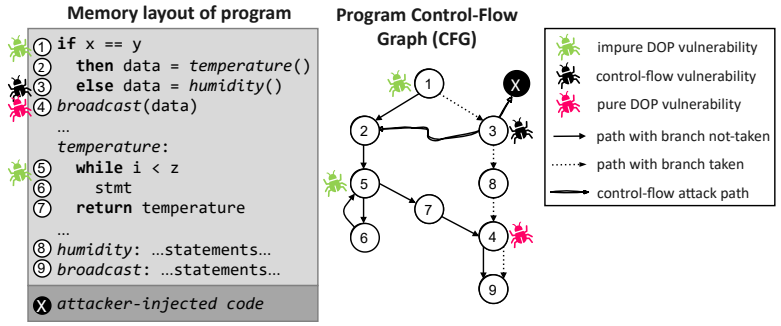


Figure 4.1: Abstract view of a program's CFG and threats.

ecution correctness to potentially untrusted verifiers, with the only requirement being that the prover can securely record and attest to the executed path, whose verification on an intermediate worker is made verifiable using VC.

4.3 Background

4.3.1 Program Composition

A compiled program's code can be represented by its Control-Flow Graph (CFG), which encapsulates all possible program executions by modeling the legal control flow between all of the program's statements. However, since not all statements affect the control flow, we typically fractionate the statements into maximal-length sequences of branch-less statements that ultimately end in a branch, jump, or predicated operation. We denote each such sequence as a basic block (BBL) and have the CFG model the control flow only between program BBLs. Let $G = (N, E)$ denote a directed graph (CFG), where nodes $n_i \in N$ correspond to BBLs and edges $e = (n_i, n_j) \in E$ denote possible transfers of control. We refer to edges corresponding to (direct and indirect) jumps and calls as forward edges and returns as back edges. We further label any node n_i an entry node ($n_{\triangleright}$) if it is unreachable (i.e., has indegree $\delta^-(n_i)$ of zero) and a final node ($n_{\blacktriangleleft}$) if it has no reachable nodes (i.e., has outdegree $\delta^+(n_i)$ of zero). (We denote with $\Delta(G)$ the maximum *outdegree* of G.) Finally, any continuous edge sequence is a legal execution path if it connects some $n_{\triangleright}$ to $n_{\blacktriangleleft}$ (denoted $n_{\triangleright} \rightsquigarrow n_{\blacktriangleleft}$).

4.3.2 Runtime Attacks

We continue with a description of major runtime attack classes that can induce harmful behavior by exploiting software vulnerabilities to corrupt a program's control and data planes. As a running example, let us consider the simple program skeleton in Fig. 4.1.
Control-based attacks. The most common attacks target a program's control plane to execute unintended code by explicitly diverting its execution path. There are essentially two variants: code injection and code reuse. With code injection, an adversary crafts and injects a payload into memory and redirects a benign program's control flow to execute the payload. As an example, consider that in Fig. 4.1 an adversary has injected

node n_X and diverted the control-flow from (n_3, n_8) to (n_3, n_X), resulting in the execution of code in n_X instead of n_8. However, being an early attacking methodology, code injection is easily defeated using common mechanisms such as Data-Execution Prevention (DEP), where executable memory regions cannot be written to during runtime. For the latter variant, however, it gets more difficult. Without injecting code, code-reuse attacks reuse existing program code to achieve some unintended behavior–using control plane maneuvers such as Return-Oriented Programming (ROP) [162] and Jump-Oriented Programming (JOP) [22].

With ROP/JOP, an adversary fabricates a new program by stitching together a chain of benign pieces of existing code (gadgets) that end in function returns (ROP) or indirect jumps or function calls (JOP). The chain is then written into memory (e.g., through a stack overflow vulnerability), where, once it is triggered (e.g., by replacing a function's return address with that of the first gadget), the gadgets execute in sequence. For example, in Fig. 4.1, the adversary launches a ROP attack diverting the control-flow from (n_3, n_8) to (n_3, n_2) to execute code in the other branch.

Non-control-data attacks. So far, we have only reflected attacks that explicitly mislead program execution down imaginary or illegal paths, which we can detect by repeating a path taken in the respective program's CFG. However, a more subtle class of attacks exists, called non-control-data attacks, which disregards a program's control plane for its data plane. Such attacks attempt to corrupt data variables to make programs yield unexpected outputs or indirectly drive program execution down unexpected or unauthorized paths without explicitly hijacking the execution path (e.g., by replacing a function's return address on the program stack). Because the path is still "legal", such attacks are difficult to detect using control-flow attestation that looks for illegal control flows.

The attacking methodology behind non-control-data attacks is the application of Data-Oriented Programming (DOP) [47, 94], which we can call impure or pure, depending on whether we influence the program execution path (impure) or alter only data variables with no effect on the path (pure). For example, in Fig. 4.1, an adversary corrupts data variables in node n_1 to change the result of the evaluation and thereby indirectly lead the execution flow down the path of her choosing. Similarly, the adversary corrupts the loop counter variable in node n_5 to modify the number of loop iterations. Both attacks are impure as their presence affects the execution path. While the attacks will remain undetected when we differentiate legal paths from illegal only by whether they conform to the program's CFG, we could detect these attacks if we already suspect a certain evaluation result in n_1 and know how often the loop in n_5 is supposed to iterate. Although such supplemental information is not always available, existing runtime attestation schemes [2, 63, 186, 62, 3, 176, 95, 125, 169, 117, 140, 141] inherently assume that verifiers have general knowledge about a program's expected behavior (e.g., loop execution and authentication information) to aid in the verification process.

However, no amount of knowledge aids the schemes [2, 63, 186, 62, 3, 176, 95, 125, 117, 140] in detecting pure DOP attacks. For example, in Fig. 4.1, we see the program blatantly transmitting data in node n_4. Here an adversary, who knows the location of sensitive data, could mount a pure DOP attack and swap the data variable's destination address with the address of the sensitive data and, thence unnoticeably exfiltrate the data.

While detecting pure DOP attacks is challenging in runtime attestation, some schemes [169, 141] make their attempts. For example, in addition to recording the execution path, [169] has provers perform local integrity verification of all program critical variables,

which have either been identified through control-dependent variable discovery or explicitly annotated by the programmer. Data integrity is continuously verified by maintaining a hash map of all critical variables, and whenever a variable is loaded, its value is compared to its previously stored entry value. Any mismatch between a variable's load and store sites signifies a DOP attack and is propagated together with the program's execution path to enable verifiers to "detect" pure DOP attacks. Contrarily, in [141], instead of entrusting provers to verify the integrity of variables, they have provers record all program data (e.g., inputs and values of accessed variables) in a log, which is delegated to knowledgeable verifiers for verification.

However, due to the difficulty of effectively verifying a program's data flow at the verifier, CFA schemes (including ours) generally disregard such attacks or assume that verification is done locally at the prover. The latter is generalized in OAT [169], where they demonstrate how we can detect DOP attacks efficiently in a prover's trust anchor by continuously verifying a program's critical variables.

4.3.3 Toward CFA in Zero-Knowledge

Section 4.3.2 explored how adversaries effectively dominate program execution by attacking its control and data planes. To detect such software attacks, attestation schemes [2, 63, 186, 62, 3, 176, 95, 125, 169, 117, 140, 141] have provers record program control and data flows, which are subsequently presented to remote verifiers for verification. However, besides the outsourcing of work to verifiers and requiring verifiers to maintain trusted reference materials, the most notable deficiency of existing schemes is the *lack of privacy* as provers must comprehensively disclose program execution details, which is unattractive when program execution should remain secret, especially in sectors with untrusted verifiers. Thus, to enable even the most security-critical and privacy-sensitive application domains to benefit from remotely verifiable program executions, we propose the novel coupling of zero-knowledge proofs to allow proving a program's execution correctness in a *privacy-respecting* manner among mutually distrusting participants.

Zero-Knowledge Proofs. First introduced by Goldwasser et al. [81] as an interactive protocol and later made noninteractive in [23], zero-knowledge proofs [106, 132] enable a prover to convince a verifier that a nondeterministic polynomial-time statement is true by demonstrating knowledge of a satisfying witness without revealing anything about the witness.

Path explosion problem. Two fundamental zero-knowledge proof system constructions are range proofs [110] and set membership proofs [144, 21], respectively. With range proofs, we can prove knowledge of a secret s by demonstrating that s belongs to the interval $[u, v)$ for arbitrary integers u and v. With set membership, we can prove that s belongs to an arbitrary set S. To create such set membership proofs, we would generally utilize some cryptographic accumulator, e.g., Merkle trees [131], or RSA-based [16]. Merkle trees, in particular, have become increasingly popular in the DLT domain, e.g., they are used in the Pour circuit of the decentralized anonymous payment scheme called ZeroCash [157] to maintain "coin commitments". However, note that to use either range proofs or set membership proofs to prove that a recorded path is legal, we would need a clearly defined interval or set during the proof system instantiation. Therefore, for us to use either approach on the worker to prove the validity of an attested execution path, we would need to know all legal execution paths beforehand, which is sometimes

impractical.

For a directed *acyclic* graph (DAG), we could enumerate all paths by performing a depth-first search. However, for a CFG, a directed graph, the set of paths is unbounded, which we can illustrate by considering the program in Fig. 4.1. For example, if the program loop is only used as a busy-wait, the set of paths grows unwieldy since any number of iterations constitutes a unique path. Moreover, as the complexity of the program increases, so does the number of paths. Therefore, we must sacrifice precision if we insist on a finite set of paths. One approach is simply transforming the CFG into a DAG by pruning all back edges. Similarly, as described in [93], we can identify all strongly connected components, i.e., maximal sets of nodes where a path exists between any two nodes in a set (e.g., the component comprising nodes n_5 and n_6 in Fig. 4.1), contract each component into a single node (e.g., fuse n_5 and n_6 in Fig. 4.1) to form a condensation graph (which is acyclic), and then consider only the paths within this condensation graph. However, both approaches are imperfect as the coarse-grained granularity of condensation or ignorance when discarding back edges would leave certain attacks undetectable.

zkSNARKs. To let resource-constrained provers attest to arbitrary execution paths without enumerating paths beforehand or sacrificing precision, we opt to use zero-knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARK) [83, 157], where we create a program that accepts a program's CFG together with any path as input and verifies that the path is legal according to the CFG. We then transform the program into a low-level arithmetic circuit C representation [20] over a finite field $\mathbb{F}$ (typically a 254-bit prime field $\mathbb{F}_p$) composed of additions and multiplications $\bmod p$. Given such a circuit, we can instantiate a zkSNARK proof system that lets provers attest to arbitrary execution paths, which can be proven correct by generating a proof π that the circuit was satisfied when executed on the attested path.

Note that in this paper, we consider what is called preprocessing zkSNARKs, where we represent the whole verifiable program to be as a single circuit of constraints, and a one-time offline (trusted) setup is needed to sample a circuit's proving and verification keys, which, afterward, must be distributed to appropriate entities in the network. Specifically, let C be our arithmetic circuit. A zkSNARK allows the worker to prove that she correctly executed C on public input x and secret input $\overline{u}$ (we use bar here to denote *secret* input), as follows. After taking C as input, a trusted party conducts a one-time setup that gives two public keys: a proving key pk and a verification key vk. The proving key pk enables any untrusted worker to produce a proof π attesting to the fact that x and $\overline{u}$ satisfied C. The non-interactive proof π is *zero knowledge* and a *proof of knowledge*. The proof reveals nothing about $\overline{u}$, but anyone can verify its correctness without prover interaction using only vk.

In total, zkSNARK schemes consist of the following three algorithms:

- $(\mathsf{pk}_\mathsf{C}, \mathsf{vk}_\mathsf{C}) \leftarrow \mathsf{KeyGen}(\mathsf{C}, 1^\lambda)$: given a circuit C and a security parameter, output pk_C and vk_C as the public proving and verification keys.
- $(y, \pi) \leftarrow \mathsf{Prove}(\mathsf{C}, \mathsf{pk}_\mathsf{C}, x, \overline{u})$: given a circuit C, proving key pk_C, public x and secret $\overline{u}$ inputs, output $y \leftarrow \mathsf{C}(x, \overline{u})$, and the proof π of the computation correctness.
- $\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{vk}_\mathsf{C}, x, y, \pi)$: given a verification key vk_C and statement (x, y), output 1 only if $y = \mathsf{C}(x, \overline{u})$.

In most constructions, C is expressed in the NP-complete languages called Rank-1-Constraint-Systems (R1CS) and Quadratic Arithmetic Programs (QAPs) [77]. In R1CS, computations are encoded as a set of conditions over its variables such that correct execution equals finding a satisfying variable assignment, whereas, with QAP, computations are instead represented as a set of quadratic equations. However, as with any VC protocol that requires the computation task to be expressed as arithmetic circuits over some field $\mathbb{F}_p$, the size of the sets (i.e., constraints) corresponds to the circuit size and determines the time needed to generate proofs. Specifically, the more constraints a circuit has, the longer it usually takes to generate proofs. Hence, in Section 4.7, when we evaluate the performance of our devised ZEKRA circuit, we will consider the number of constraints as the primary performance metric of our solution.

4.4 System and Threat Model

Before we describe the technical details of the ZEKRA protocol, we briefly present the considered setting and assumptions concerning the protocol participants.

4.4.1 System Model

We consider a network with four types of entities:

1. **Prover** is an untrusted and underpowered device equipped with a minimal trust anchor capable of only tracing and authenticating a program's execution. Note that these capabilities (tracing and cryptographic functions for signing a program's execution path) constitute the minimum trusted computing base for guaranteeing the security of the attestation and thus restrict as much as possible the influence and performance footprint of the underlying trust anchor on the normal execution of the prover's host operating system. See Section 4.4.4.2 for further clarification and justification on the underlying trust assumptions concerning the prover. However, note that the choice of tracing via (i) interfacing with the CPU's pipeline [63, 186, 62], or (ii) having instrumented programs, stored in DEP-enabled memory, self-report control-flow transfers [2, 3, 176, 169], is considered complementary to our work.

2. **Verifier** is an untrusted device wishing to check the correctness of a program (or part thereof) executed on the prover.

3. **Worker** is a semi-untrusted and computationally capable device that generates zk-SNARK proofs for convincing untrusted verifiers about the correctness of a prover's attested execution paths without disclosing any secret inputs (e.g., the attested execution path or reference materials). See Section 4.4.4.1 for further clarification on the worker's role.

4. **Network operator** is trusted party that executes the KeyGen algorithm (see Section 4.3.3) and equips the protocol participants with necessary key materials. Note, however, that while we consider the network operator as a central trusted entity

who generates the cryptographic keys associated with a specific circuit, in practice, a secure multi-party sampling protocol would replace the zkSNARK circuit's trusted setup ceremony [27].

4.4.2 Adversarial Model

We assume a strong software adversary, who, on the prover, exploits a severe software vulnerability to mount control-flow attacks to divert the attested program's execution. We then assume that the semi-dishonest worker is colluding with the prover in an attempt to convince the verifier that the attested program was executed correctly when in reality, it was not. Finally, we consider an untrusted verifier that attempts to infer details about the prover's program (without colluding with the prover's or worker's adversaries). Note, however, that if the verifier also colludes, this is limited to violating the protocol's privacy objective.

4.4.3 Protocol Objectives

Our protocol's objectives are threefold: (i) verifiers always reject a proof unless the prover executed the expected program (or segment thereof) correctly and no control-flow attack was present, (ii) any attempt by the worker to manipulate inputs during proof generation results in a rejection, and (iii) verifiers neither require nor learn any program details from the verification process.

4.4.4 Trust Assumptions

This section addresses frequently asked questions about our protocol which we describe in Section 4.5. While some content repeats what is already stated in the paper, we reiterate some important points that a reader may miss.

4.4.4.1 On the Trustworthiness of the Worker

Our scheme considers a resource-constrained prover device capable of securely recording and signing the execution path taken by a program during runtime. While the prover wants to assure a verifier that the program was executed correctly (i.e., in the absence of control-flow attacks), the prover wants to keep the path and program details *private*. Therefore, the prover outsources the signed execution path to a resourceful worker who produces a zero-knowledge proof of the path's correctness which is publicly verifiable and does not reveal the execution path. However, since the prover gives the secret inputs (i.e., the recorded execution path and blinding factors) to the worker (who also knows all reference materials) for producing the proof, it must trust that the worker keeps the inputs secret. In other words, we trust the worker regarding the posterior privacy of the secret circuit inputs to satisfy our protocol's privacy goals as described in Section 4.4. However, the worker is *not a trusted party* since the worker can attempt to cheat in producing the proof (e.g., by tampering with the inputs), which is why we require verifiable computation to let verifiers detect such *dishonest* behavior. Therefore, since the worker is trusted regarding posterior input privacy but untrusted regarding proof generation, we refer to the worker as semi-dishonest. For future work, we note the possibility of further

weakening our trust assumptions (on the worker) by hiding the secret inputs, e.g., by employing multiparty computation techniques based on Shamir's secret sharing [159] in networks comprising multiple workers.

4.4.4.2 Minimal Trusted Computing Base

As mentioned in Section 4.4, a prover is a low-end embedded device with limited resource capabilities. Hence, it is highly desirable to restrict the influence of the underlying trusted component (i.e., trust anchor) on the normal execution of the host, which is a common barrier affecting the generality and applicability of existing remote attestation schemes to safety-critical systems [39]. Thus, ZEKRA's design choice is to follow a minimalist attestation approach [73], assuming the existence of a root-of-trust with only those properties needed to attain remote attestation services. These include recording a program's execution path and cryptographic functions for signing the recorded execution path to guarantee origin authentication. Note that for the latter, the resource overhead is mainly determined by the hashing since the signing operation is independent of the execution path size, as only a fixed-size hash is signed. For the former, one natural way to extract an executing program's execution path is to equip devices with tracing capabilities, e.g., by leveraging existing processor hardware features and commonly-used IP blocks, as done by the tracer proposed in LO-FAT [63]. Note that while tracing program execution is relatively efficient with minimal perturbation, interpreting (i.e., translating in our case) raw memory addresses and verifying the recorded execution path's correctness is complex as it relies on additional trusted reference materials. ZEKRA decouples these two functionalities (recording and translating/verification). Thus, ZEKRA enables a minimal trust anchor that only needs recording support without additional decoding capabilities. Specifically, the trust anchor only records raw traces (capturing sequences of memory addresses visited), which are then sent to the worker for the more complex and program-specific task of convincing the verifier that the execution path was correct according to a specific program's trusted reference materials. Resolving this inherent link between tracing and program-specific decoding, ZEKRA supports devices with continuous (non-intrusive) tracing capabilities (e.g., ARM Coresight) which offer negligible impact on the performance of the programs executing in the normal world. Specifically, note that recording execution paths is an efficient and program-agnostic process and, thus, can easily fit inside a small trust anchor. Especially since only the hash of the recorded execution path, which is being accumulated during program execution, needs to be securely stored, while the path itself can reside in unprotected memory, as done in most CFA schemes. However, since the verification is program-dependent, it would require the trust anchor to maintain all trusted reference materials (i.e., CFGs and translator mappings, which can grow large for complex binaries, and also all possible entry/exit node pairs, including semantical information to determine which pair to consider for each attestation) in *secure memory* for each of the programs (and attestable segments thereof) offered by a prover. We would also require additional mechanisms on provers to guarantee attestation freshness to verifiers. Furthermore, the trust anchor's responsibility becomes even more complex if we consider embedded systems with the possibility of over-the-air updates since the trust anchor would need additional logic to set up *secure communication channels* with trusted authorities for updating its trusted reference materials. Therefore, it becomes clear that reconciling the needs of safety-critical applications

and RA security requirements through a minimal architecture for the underlying trust anchor enables ZEKRA to support practical remote attestation with minimal requirements over the prover's computational resources. In this process, we remain agnostic regarding the underlying hardware by making the fewest possible assumptions about specific devices. We believe that the outcome of our minimalistic design is valuable, as it pushes towards a lightweight blueprint of remote attestation that can be realized on a wide range of low-end devices, with minimal modifications and assumptions on required secure hardware.

4.5 The ZEKRA Protocol

The ZEKRA protocol augments existing CFA schemes by encasing the rigorous task of verifying that a program's execution path is benign according to the program's CFG in a circuit, which can be proven without disclosing the executed path or program CFG (zero-knowledge property).

4.5.1 CFG Conformance

For the worker to convince an untrusted verifier that an execution path $\mathcal{EP}$ is benign according to the reference program's CFG in zero-knowledge, she must prove the statement "I have successfully verified that $\overline{\mathcal{EP}}$ is a legal path in $\overline{CFG}$, which began at node $n_{\triangleright}$ and ended at node $n_{\blacktriangleleft}$, where $\overline{CFG}$ is the preimage of h_1". To prove this statement, we embed it in a circuit C, which we refer to as the ZEKRA circuit, where **overlined** variables denote secret inputs to the circuit as described in Section 4.3.3, i.e., we have the secret $\overline{u} = \{\overline{\mathcal{EP}}, \overline{CFG}\}$ and public $x = \{n_{\triangleright}, n_{\blacktriangleleft}, h_1\}$ inputs, respectively.

To allow verifiers to verify whether the correct program's CFG was considered for a given proof, we assume verifiers know the digest of the program's CFG as a reference value, $h_1 = \mathsf{H}(CFG \| r_1^{\lambda})$, where r_1 is some sufficiently-long random padding (blinding factor) added to the CFG preimage to protect against hammering and linkage attacks and λ denotes the security parameter. (In our case, we consider $\lambda = 254$, corresponding to a 254-bit prime field $\mathbb{F}_p$.) Thus, given a valid proof π over C and public inputs x used in the proof generation, verifiers can verify that the correct CFG was considered by checking that the public input digest matches the expected reference value. However, verifiers cannot infer anything about the CFG preimage, which was supplied as a secret input.

Similarly, to let verifiers determine whether the secret execution path supplied (attested) by the prover also connects the expected CFG nodes, e.g., that it entered as expected at node n_1 and exited at node n_9 in Fig. 4.1 (thus marking a successful execution), we grant verifiers knowledge about the CFG's contextually relevant entry node $n_{\triangleright}$ and exit node $n_{\blacktriangleleft}$, respectively. These nodes are public inputs to the ZEKRA circuit to let verifiers observe them and are used internally to verify the start and end of the supplied execution path. To simplify the discussion, we assume that each CFG has a unique entry node, $n_{\triangleright}$, and a unique exit node, $n_{\blacktriangleleft}$. However, the procedure is the same regardless of the considered granularity (e.g., program level or function level), where a CFG might have multiple legal entries or exit nodes. Note that since the interlinking execution path $n_{\triangleright} \rightsquigarrow n_{\blacktriangleleft}$ remains secret, the verifier cannot infer anything about the execution path from observing the endpoints.

Furthermore, note that here $n_{\triangleright}$ and $n_{\blacktriangleleft}$ do not refer to the actual memory addresses of the corresponding BBLs in the program CFG but to numeric labels that have been assigned to the corresponding nodes in the CFG. Specifically, because we must traverse the CFG in the ZEKRA circuit, we must represent the CFG as a traversable data structure, and using the nodes themselves to index the structure allows for more optimized lookups. (We discuss how we represent the CFG in Section 4.5.4.) However, since the prover will record and attest to the raw execution path, which includes the actual memory addresses, we assume a mapping $\mathcal{M}$ to help the ZEKRA circuit first translate the recorded addresses into their corresponding numerical label representation. The mapping is simply a list of the possible memory addresses (i.e., nodes in the program CFG), where the index of an address denotes its numeric label.[1] Then, similar to the CFG, the circuit accepts $\mathcal{M}$ as secret input and $h_3 = \mathsf{H}(\mathcal{M} \| r_3^{\lambda})$ as public input, where r_3 is the random padding (blinding factor) added to $\mathcal{M}$'s preimage, which the verifier (who knows only $\mathsf{H}(\mathcal{M} \| r_3^{\lambda})$ as a reference value) can verify by observing the circuit's public inputs.

4.5.2 Path Authenticity

Note that anyone knowledgeable about the program or its CFG can identify paths that will satisfy the ZEKRA circuit. Thus, to convince verifiers that the secret execution path for a particular proof was recorded on the prover and not produced by someone else, we assume that each prover's trusted tracer has a certified asymmetric key pair $\{\mathsf{tpk}, \mathsf{tsk}\}$, where tpk denotes the public key, and tsk denotes the secret key, respectively. It follows that verifiers must know a prover's public key to verify the authenticity of attestation materials signed using that prover's secret key.

One method of convincing the verifier about the path's authenticity is requiring a prover to sign the recorded execution path $\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{H}(\mathcal{EP}), \mathsf{tsk})$, have the circuit accept Sig and $\mathsf{H}(\mathcal{EP})$ as secret inputs and tpk as a public input, and then have the circuit use tpk to verify internally that Sig is a valid signature over $\mathsf{H}(\mathcal{EP})$ and $\mathsf{H}(\mathcal{EP})$ is the correct digest of $\mathcal{EP}$. Then the verifier can verify that the correct prover authenticated the execution path by checking if the correct tpk was supplied. However, the problem is that signature verification is expensive in terms of circuit size since most algebraic signature schemes are not compactly expressed over a field $\mathbb{F}_p$. For example, expressing the RSA algorithm, which heavily relies on modular exponentiation and long integer arithmetic, yields close to 90K constraints [114], even considering a hardcoded modulus and considerable optimizations. While there exist techniques to reduce the complexity, e.g., by using a small public key exponent [137], there are currently, to the best of our knowledge, no efficient general-purpose signature schemes for circuits.

Another method is to have the prover prove possession of the secret key behind its public key. For example, assuming that the RSA cryptosystem is considered, we could include the substatement "I know $\overline{p}$ and $\overline{q}$, where $\overline{p} \times \overline{q} = n$" as part of the ZEKRA circuit's underpinning statement since knowledge of $\overline{p}$ and $\overline{q}$ for some public key modulus n proves possession of the secret key. However, this would require the prover herself to generate the proof, which is unsatisfactory, especially since CFA schemes generally

[1] Another benefit of keeping the CFG representation abstract inside the circuit is that we are not limiting what is being mapped. For example, instead of mapping BBL addresses, we could also include the hashes of the executed code as done in ATRIUM [186]. We show how this extension has negligible performance impact in Section 4.7.2.

consider resource-constrained or heavily embedded devices and have the prover only be concerned with tracing the program before outsourcing the signed execution path to the verifier. Therefore, without complicating the prover, we must design the ZEKRA circuit with the intention of the proof generation being outsourced to workers.

For the third method of proving path authenticity, which we opted for in our current version, signature verification is performed outside the circuit, as inspired by [137]. The idea is for the circuit to accept $h_2 = \mathsf{H}(\mathcal{EP} \| nce^\lambda \| r_2^\lambda)$ as public input, where nce^λ is a fresh nonce generated by the verifier to ensure freshness, and r_2^λ is some random padding (blinding factor) generated and added to the execution path by the prover. The nonce is given as public input to the circuit to let verifiers ascertain freshness while the blinding factor is kept secret. The circuit then verifies internally that the secret execution path $\overline{\mathcal{EP}}$, padded with the nonce and blinding factor, is indeed the correct preimage of h_2. As in the first method, the prover also signs the recorded execution path $\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{H}(\mathcal{EP} \| nce^\lambda \| r_2^\lambda), \mathsf{tsk})$. The worker's proof and prover's signature are then given to the verifier, who verifies that the public digest h_2 used in the proof generation matches the prover's signed digest.

We give more details on the ZEKRA circuit in Section 4.6. Let us first bring it all together and clarify the overall protocol.

4.5.3 The Protocol

Fig. 4.2 shows a prover and a verifier engaging in the ZEKRA protocol after the network operator has performed the required one-time setup of executing KeyGen for the ZEKRA circuit C for some *sound* zkSNARK proof system and equipping participants with the appropriate cryptographic materials. The protocol then executes as follows.

To ensure freshness, the verifier challenges the prover with a nonce *nce* and a reference @P to the program or procedure she wants to be executed and attested. In practice, the attested region is only a subset of the entire program [169, 3], e.g., a security-critical function or code section. The prover then executes the program while its trusted tracer chronologically traces the executed path $\mathcal{EP}$ when executing the region to be attested. Once the program concludes, the trusted tracer hashes and signs the execution path padded with the verifier nonce and some freshly sampled blinding factor. The signed digest is then given to the worker with its secret ingredients, where the worker is responsible for computing proof over the ZEKRA circuit with the execution path as secret input. Before generating a proof, however, the worker first converts the execution path into its numerical label representation $\mathcal{L}$ using the address-to-label mapping $\mathcal{M}$, which the circuit can then verify to be done correctly instead of having to perform the computationally-intensive conversion task (further optimizations are discussed in Section 4.6). The worker then computes a zkSNARK proof by executing Prove and passing in as secret inputs: the attested program's reference materials (i.e., the CFG and address-to-label mapping $\mathcal{M}$, along with their blinding factors), the attested execution path (including its blinding factor), and the numerical representation of the attested execution path $\mathcal{L}$. As public input, the worker passes in the digests of the CFG, mapping $\mathcal{M}$, and the execution path, together with the relevant entry and exit nodes and the verifier nonce.

Once the proof is generated, the proof, its public inputs, and the prover's signature over the execution path commitment digest are sent to the verifier. (Note that the worker is shown to transmit the data directly back to the verifier to simplify the message ex-

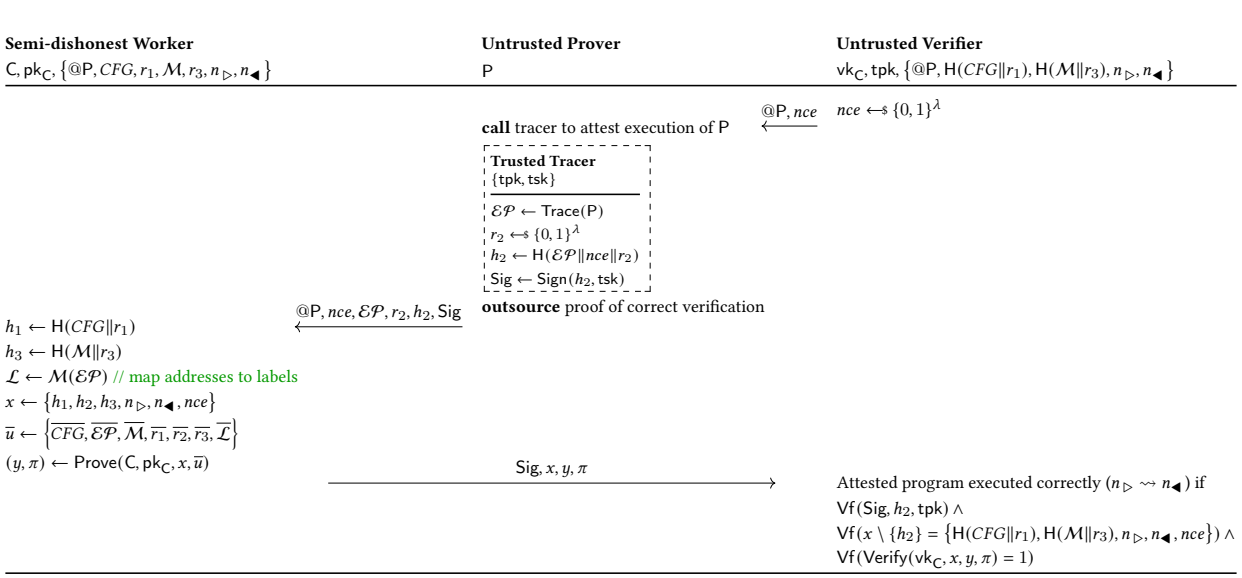


Figure 4.2: The ZEKRA protocol, where, upon request, a prover attests to the execution of a program before outsourcing the task of convincing the untrusted verifier about the execution's correctness in zero-knowledge to a semi-dishonest worker.

changes.) Finally, assuming the soundness of the considered proof system, the verifier is convinced that the intended program was executed correctly, in the absence of any control-flow attacks, on the prover if: (i) the proof is satisfied under the circuit's verification key, (ii) the execution path commitment supplied as public input to the proof generation–whose corresponding preimage ingredients were supplied as secret inputs and verified internally in the circuit to hash to the public commitment digest–was signed by the prover, (iii) the CFG and mapping $\mathcal{M}$ of the intended program were considered, and (iv) the correct start and end nodes were visited. If these criteria are satisfied, the verifier is convinced that the intended program (or segment thereof) was executed correctly on the prover.

4.5.4 Building Blocks

Before explaining our circuit's design, we must understand how we represent and work with execution paths and program CFGs.

4.5.4.1 Tracing

We consider a device which is equipped with a tracer capable of recording all control-flow events during a program's execution. This tracer can be realized either in software [2, 3, 176, 169, 140, 141] or hardware (e.g., Intel Processor Trace, ARM CoreSight, or custom hardware extensions [63, 186, 62]). Regardless, when chronologically tracing a program's execution, we assume that each execution path $\mathcal{EP}$ (of some length E) is marshalled in the form of a sequence of control-flow transitions: $\mathcal{EP} = (t_1, t_2, \ldots, t_n)$,

where each transition $t_i = (jmpkind, n_{dst}, n_{ret})$ includes a 2-bit identifier for the type of transition, i.e., whether the transition was caused by a jump, function call, or function return, the destination addresss (supposed entry of the target BBL), and a return node n_{ret}, which for calls points to a BBL where the callee function should return. Note that knowing the transition type enables shadow stack emulation for ensuring back edge integrity, which we describe in Section 4.6.

4.5.4.2 Control-Flow Graph Representation

A core part of the ZEKRA circuit is how we represent and traverse a CFG. To model the set of legal transitions, we can either encode the set of legal edges as an adjacency matrix or adjacency list. In the matrix format, we can represent a digraph $G = (N, E)$ as a two-dimensional array M of size $N \times N$, where a slot $M(n_i)(n_j) = 1$ indicates that an edge exists from node n_i to node n_j. The advantage of the matrix is that we can determine in $O(1)$ whether an edge exists from n_i to n_j. However, since the matrix has space complexity of $O(|N|^2)$ it requires a prohibitively large data structure to be expressed in an arithmetic circuit. Contrarily, in the adjacency list, we only store a node's reachable neighbors, which reduces the space complexity to $O(|N| + |E|)$ but increases query time complexity to $O(|N|)$. While the space complexity is better than the matrix, it can become expensive for dense areas in a CFG where a node might have many reachable neighbors (e.g., a program switch with a large jump table).

To reduce the space complexity even further, we leverage the idea behind the **IndexedBitArrayEdges** representation as proposed in [123], which takes advantage of the concentration of edges in specific areas of the adjacency matrix. With this encoded representation we use a single byte to represent eight possible out-neighbors. Using an array, we construct a data structure of $(bucket + 8)$-bit elements, one for each neighbor label interval with the same quotient when divided by 8. Each element's first $bucket - 8$ bits represents the quotient (bucket), while the last byte serves as a set of 8 flags indicating whether each possible edge exists in this interval. Note that it follows that $bucket$ must at minimum be $\lfloor \log_2((N-1)/8) \rfloor + 1$ bits for us to represent all possible quotients of CFGs with N nodes.

Since the circuit performs execution path verification on CFGs with all nodes relabeled using consecutive integers (where the relabeling is reflected in the mapping $\mathcal{M}$), we can represent any N-node abstract CFG as a single-dimensional adjacency list of size N, whose indices correspond to CFG nodes and contain the indexed node's encoded neighbors. Also, since the maximum label is $N - 1$ when numbered from 0, the circuit only needs to allocate $\lfloor \log_2(N-1) \rfloor + 1$ bits per label to represent the execution path.

To better understand how we apply the encoding, assume that a node has the following set of neighbors: $\{288, 289, 290, 291, 292, 293, 294, 614\}$. We can group the neighbors in two sets: $\{288, 289, 290, 291, 292, 293, 294\}$, $\{614\}$, where the first set shares bucket 36 when divided by 8, and the second set share the bucket 76. We then iteratively store the bucket of each neighbor set as the first $\lfloor \log_2((N-1)/8) \rfloor + 1$ bits and the remainders ($rems$) as the neighboring byte. We refer to each such $(bucket, rems)$ pair as a level and use ℓ to denote the maximum levels of the encoded adjacency list. Note, however, that the number of levels needed depends on how the adjacency list's labels are arranged. For example, in our example, we require two levels to accurately represent all eight neighbors, where the 0th to the 6th bit of the first level's $rems$ are set to 1 to indicate the first

seven neighbors. However, if we could rearrange the numerical adjacency list such that the considered node's neighbors all shared the same quotient, it would only need one level. (We defer this graph optimization problem and other CFG reduction/compression methods as they complement our work.)

Finally, given an encoded adjacency list $\mathcal{AL}$, we determine if node n_j is a valid neighbor of node n_i by verifying that there exists some $(bucket, rems)$ pair in $\mathcal{AL}$, such that $\lfloor n_j/8 \rfloor = bucket \wedge rems[n_j \bmod 8] = 1$, where $rems[n_j \bmod 8]$ denotes a bit in $rems$ at position $n_j \bmod 8$. In other words, we check that the destination node's bucket exists and the corresponding remainder bit is on.

4.5.4.3 Hashing

When selecting a suitable hashing function H to utilize in our circuit for hashing the adjacency list $\mathcal{AL}$, attested execution path $\mathcal{EP}$, and address-to-label mapping $\mathcal{M}$, the deciding factor is how inexpensively it can be expressed in an arithmetic circuit. Fortunately, several hash functions have been proposed due to increased attention to circuit-based zero-knowledge proofs. The most recent, which currently offers the best performance, is the cryptographic permutation function called Poseidon [82], which takes a set of elements of a certain field $\mathbb{F}$, called scalars, as inputs and outputs one scalar. The number of inputs determines the width $w = r + c$ of the internal state, where r and c are called the *rate* and *capacity* of the permutation function. Setting the capacity to one field element in a 254-bit field $\mathbb{F}$ offers a 128-bit security level and using a rate (arity) of 4, the hashing function essentially corresponds to a 4-to-1 compression function.

In our case, we configured Poseidon-128 with a width of 9 to achieve a rate of eight field elements per call, which allows us to walk over the data structures during hashing more quickly. Since the considered data structures can be arbitrarily/selectively large, we used the proposed Poseidon constant-length sponge-based construction [82]. Let $X = (x_1, \ldots, x_m)$ refer to an execution path $\mathcal{EP} = (t_1, \ldots, t_m)$ of l-bit transitions, an adjacency list $\mathcal{AL} = (e_1, \ldots, e_m)$ of l-bit encoded neighbor entries, or a mapping $\mathcal{M} = (a_1, \ldots, a_m)$ of l-bit entries, respectively. We then hash X into a single scalar (i.e., field element) as follows:

1. Compress X by successively fitting $\lfloor (\mathbb{F}\text{'s bitwidth})/l \rfloor$ l-bit values from X into one field element and storing the resulting value in X'. Let t denote the size of X'.
2. Pad X' with zero elements up to the multiple of 8, then split it into chunks $w_1, w_2, \ldots, w_{\lceil t/8 \rceil}$, each containing 8 scalars.
3. Apply the permutation function Poseidon to the capacity element and the first chunk.

   $$(h_1^1, h_1^2, \ldots, h_1^9) \leftarrow \textsc{Poseidon}(len \times 2^{64} + (o-1), w_1)$$

   (Note that the capacity field is set to $len \times 2^{64} + (o-1)$, where len is the input length and o is the output length (usually $o = 1$). In our case the input length is 8 field elements, and the output length is 1 field element.)
4. Until no more chunks are left, apply the permutation:

   $$(h_i^1, h_i^2, \ldots, h_i^9) \leftarrow \textsc{Poseidon}(h_{i-1}^1, h_{i-1}^2 + w_i^1, \ldots, h_{i-1}^9 + w_i^8)$$

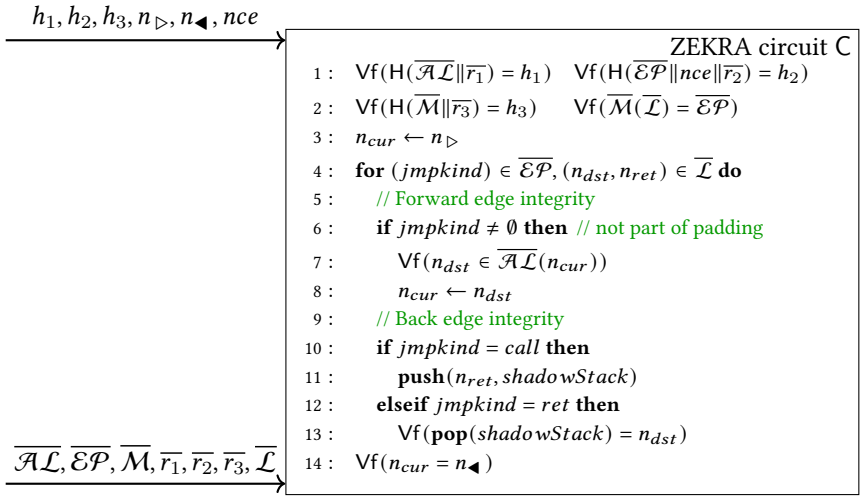


Figure 4.3: High-level algorithm of the outsourceable ZEKRA circuit, with its secret (top left) and public (bottom left) inputs.

5. Output o output elements from the rate part of the state, i.e., in our case, the digest of X is the second element:

$$\mathsf{H}(X) = h_{\lceil t/8 \rceil}(2)$$

We utilize this sponge construction in our circuit to verify the considered secret preimages against their corresponding public digests before relying on them to accurately report on the expected program's reference materials and the execution path. Thus, the public digests must also be computed similarly on the outside.

4.6 On the Design of the ZEKRA Circuit

Before detailing our design choices, we briefly overview the different circuit components/gadgets. The high-level algorithm of the ZEKRA circuit is presented in Fig. 4.3 along with its $\overline{\text{secret}}$ and public inputs. Note that we have replaced the CFG in Fig. 4.2 with the encoded adjacency list $\mathcal{AL}$ of size N. After verifying the preimages and the correctness of the translation of the attested execution path $\mathcal{EP}$ of size E into its label representation $\mathcal{L}$, the circuit traverses the label version of the execution path to verify its legality, according to the adjacency list and the expected entry and exit nodes.

Forward edge integrity. To verify that the execution path satisfies forward edge integrity, we maintain a state variable n_{cur} (initialized to $n_{\triangleright}$) as we walk the execution path to verify its legality, i.e., that it is a continuous sequence of transitions that only flow through neighboring (adjacent) nodes, as follows. For each transition t_i in the execution path, we consult n_{cur}'s neighbors from the adjacency list and verify that t_i's destination node n_{dst} is indeed listed as a valid neighbor, and if so, we update n_{cur} to n_{dst}. Thus, by verifying that it reached the final node $n_{\blacktriangleleft}$ on exit, we are certain, by transitivity, that

the execution path is legal and also correctly connects the expected endpoints. However, note that while a node can have several reachable and equally valid neighbors in the forward direction, this is not true for backward edges. Specifically, when a function returns, *it should only return to where the caller intended.*

Back edge integrity. Therefore, to ensure exact back edge integrity, we consider, similar to other CFA schemes [176, 169], the use of a shadow stack (of some depth D) to simulate the traditional program stack as we walk the execution path. For function calls, we push the return node n_{ret} on the stack, and for returns, we verify that n_{dst} indeed is the stack's topmost element.

4.6.1 Circuit Design Challenges

Besides the challenges that have already been described, e.g., enabling execution path authenticity in Section 4.5, representing the CFG as a space-efficient encoded adjacency list, and selecting a circuit-friendly hashing algorithm in Section 4.5.4, we are missing the actual execution path traversal.

Due to the complexity of expressing computations as circuits, many circuit construction tools include programmable interfaces and compilers to optimize the translation of computations expressed in a higher-level language into circuits [67, 98, 114]. However, while the circuit compilers let us not worry about the low-level wiring process, they are not as mature as standard program compilers, and there are thus many employable techniques to further reduce the number of constraints needed to express a particular program. Specifically, note that the complexity of any program in terms of the number of constraints it compiles down to is the sum of the cost of expressing all statements, all loop iterations, and accounting for all branches. To illustrate what we mean by cost, first imagine the ZEKRA algorithm in Fig. 4.3 as a standard program that we want to transform into an arithmetic circuit. When the code is fed to a circuit compiler, it will flatten out the code by unrolling the loop to the worst-case number of iterations (as determined by the size of the execution path data structure, set to some value E) while taking each branch of each conditional statement into account for each iteration. Then, the compiler will convert the code to single static assignments, which are then transformed into one or more constraints [28, 180]. The result is a concise set of constraints (or equations) that is satisfied only when all variables hold all equations simultaneously. For a simple statement "$a = b$", the equivalent constraint set might consist of the constraints representing a, the constraints representing b, and an additional constraint relating the outputs of a and b [161]. Therefore, operations that cannot easily be expressed, e.g., modulo, division, exponentiation, "bit twiddling", and random-memory access, are *costly.* Fortunately, as stated in [114], an essential strategy to optimize complexity is the observation that it usually suffices and is generally cheaper to have a circuit verify a computation's correctness instead of computing the function in the forward direction. For example, $y = x/a$ can be verified more efficiently by checking that $ya = x$ rather than computing the division in the forward direction. Similarly, when working with modular arithmetic, the joint statements $r \equiv x \bmod q$, $y = \lfloor x/q \rfloor$, where q is the modulus, can be verified more efficiently by checking that $yq + r = x \wedge r < q$.

4.6.1.1 Random-Accessed Memory

While sequentially walking the execution path incurs a negligible cost, verifying each transition's destination node n_{dst} according to the n_{cur}th entry in $\mathcal{AL}$ requires random memory access, which is expensive since no information is known about which element is being accessed during compile-time. The same applies to verifying that the numeric execution path $\mathcal{L}$ is a correct translation of $\mathcal{EP}$ according to $\mathcal{M}$, as shown in (4.1).

$$\begin{aligned}\forall f(\forall i \in \{0 \dots E\} : &\mathcal{M}(\mathcal{L}(i)(dst)) = \mathcal{EP}(i)(dst) \wedge \\ &\mathcal{M}(\mathcal{L}(i)(ret)) = \mathcal{EP}(i)(ret))\end{aligned} \tag{4.1}$$

The typical, naive approach for realizing dynamic memory is performing a linear scan of the entire array memory to select one element for each memory access, which results in $O(kn)$ cost for making k memory accesses where n is the array's total memory size. Another approach, supported by recent compilers [180, 114], involves a permutation network with complexity $O((n+k)(\log(n+k)))$. However, seeing how the smart memory recently proposed in xJsnark [114] has outperformed the permutation network with a complexity of $O(k\sqrt{n})$, promising $2\sqrt{n} + \log_2 n$ constraints per access, we opted to express the current version of ZEKRA in xJsnark.

4.6.1.2 Representing the Adjacency List

Note that xJsnark currently only supports one-dimensional arrays with its smart memory type. Therefore, we have the circuit accept the adjacency list $\mathcal{AL} = (neighbors_1, \dots, neighbors_m)$ as a single, node label indexable array of field elements, whose sizes correspond to the size of our finite prime field $\mathbb{F}_p$ (in our case 254 bits), i.e., each node n_i's $(bucket, rems)$ pairs are consecutively concatenated into one 254-bit field element $neighbors_i = p_1 \| p_2 \| \dots \| p_\ell$. However, the challenge with the concatenation is determining whether a specific destination node n_{dst} is a neighbor of some node n_i when $neighbors_i$ is given as a numeric value.

While we could enforce a specific type on $\mathcal{AL}$'s elements, e.g., that they be unsigned integers instead of native field elements, allowing us to iterate over the $(bucket, rems)$ pairs using either bitwise or division/exponentiation operations, recall that these operations are expensive. Furthermore, by restricting input values to n-bit unsigned integers, the circuit must verify that the supplied values are n bits while continuously ensuring that any operation on the integers results in a value that fits within that range, essentially resulting in $n+2$ additional constraints for each integer [114]. Contrarily, by keeping the input values as native field elements, they are guaranteed to remain within a certain bitwidth.

Therefore, instead of retrieving each $(bucket, rems)$ in the circuit, we apply the power of SNARK verification by accepting (as secret input) another two-dimensional (non-smart) array $\mathcal{AL}'$ of size $E \times 2\ell$, containing the pairs already split into separate elements. However, contrary to $\mathcal{AL}$, we access $\mathcal{AL}'$ sequentially, which is made possible by requiring $\mathcal{AL}'$ to be ordered with relevance to the transitions, i.e., the ith index of $\mathcal{AL}'$ contains n_{cur}'s pre-split $(bucket, rems)$ pairs at transition i, which is easily arranged by the worker. However, before relying on $\mathcal{AL}'(i)$, the circuit first verifies that $\mathcal{AL}'(i)$ correctly represents $\mathcal{AL}(n_{cur})$ by checking that $\mathcal{AL}'(i)$ computes to $\mathcal{AL}(n_{cur})$ as shown

in (4.2).

$$\mathsf{Vf}\Big(\Big(\sum_{j=0,2,\ldots}^{2\ell-1} \mathcal{AL}'(i)(j+1) \times 2^{\lfloor j/2 \rfloor (bucket \text{ bitwidth}+8)+8} + \mathcal{AL}'(i)(j) \times 2^{\lfloor j/2 \rfloor (bucket \text{ bitwidth}+8)}\Big) = \mathcal{AL}(n_{cur})\Big) \tag{4.2}$$

We continue by detailing how we can apply further optimizations to efficiently check whether a particular neighbor exists in the adjacency list in Section 4.6.1.3, and then we summarize the final circuit design in Section 4.6.2.

4.6.1.3 *Efficient Graph Traversal in a Linear System of Equations*

While we can use $\mathcal{AL}'$ to efficiently (i.e., inexpensively) access n_{cur}'s encoded neighbors as we iterate over the execution path as described in Section 4.6.1, which operations we use to check whether a particular neighbor exists directly impacts the circuit's performance. For example, using the mathematical equations outlined in Section 4.5.4, we can effectively determine if n_{dst} is listed as n_{cur}'s neighbor at the execution path's ith transition by checking that $\mathcal{AL}'(i)(l) = \lfloor n_{dst}/8 \rfloor$ and $\mathcal{AL}'(i)(l+1)[n_{dst} \bmod 8] = 1$ hold for some level j, where j is an even number and $0 \leq j < 2\ell - 1$. However, while the integer division and modulo operations, $\lfloor n_{dst}/8 \rfloor$ and $n_{dst} \bmod 8$, are not exceptionally expensive due to the constant divisor (arbitrary modulo operations, however, are expensive, especially on prime fields as they require a range check: $a = qb + r, r < b$), the latter equation assumes bitwise operations to access bit $(n_{dst} \bmod 8)$ of $rems$, which can become prohibitively expensive if done repeatedly. Specifically, to access individual bits of an n-bit integer-carrying wire, e.g., $rems$, in the circuit, we require, at a minimum, a similar number of constraints to cover each bit of the wire. Thus, to alleviate the need to access individual bits using bitwise operations, we transform the equations into pure algebraic expressions as shown in (4.3), where the left-hand side of the latter equation mathematically accesses the value of the $(n_{dst} \bmod 8)$th bit of $rems$ by first pruning the bottom $(n_{dst} \bmod 8)$ bits using integer division and then cherry-picking the low order bit of the result using modulo.

$$\begin{gathered} \lfloor n_{dst}/8 \rfloor = bucket \\ \left\lfloor rems/2^{n_{dst} \bmod 8} \right\rfloor \bmod 2 = \begin{cases} 0, & \text{bit is unset} \\ 1, & \text{bit is set} \end{cases} \end{gathered} \tag{4.3}$$

Thus, if we find a pair $(bucket, rems) \in \mathcal{AL}'(i)(n_{cur})$ satisfying (4.3) for some transition i's destination node n_{dst}, we know that n_{dst} is a valid neighbor of n_{cur}, i.e., $n_{cur} \rightsquigarrow n_{dst}$ is legal. The remaining challenge, however, is that the components of the latter equation in (4.3) require expensive *variable* integer division and *variable* exponentiation.

However, applying the power of SNARK verification, rather than doing the variable integer division $\lfloor rems/2^{n_{dst} \bmod 8} \rfloor$ and variable exponentiation $e = 2^{n_{dst} \bmod 8}$ in the circuit, we make the prover supply the answer of the integer division (the quotient, q_1, and remainder r_1) and exponentiation e, and then in the circuit we verify that $eq_1 + r_1 = rems$, where $r_1 < e$, and e is the expected product of $2^{n_{dst} \bmod 8}$. Note that we can efficiently check whether e is the expected product since we work with a constant divisor of 8 and can therefore create a fixed sequence of all possible products: $P = \left(2^0, \ldots, 2^7\right)$.

Table 4.1: Auxiliary variables (hints) that allow the circuit to efficiently verify that a transition's destination node n_{dst} exists in the current node n_{cur}'s encoded neighbor list. Basically, for some $(bucket, rems) \in \mathcal{AL}'(n_{cur})$:

Worker computes	Variable in π_{exists}	Circuit verifies
$2^{n_{dst}} \bmod 8$	e	$e = 2^{n_{dst}} \bmod 8$
$\lfloor rems/e \rfloor$	q_1	$eq_1 + r_1 = rems$
$rems \bmod e$	r_1	$r_1 < e$
$\lfloor q_1/2 \rfloor$	q_2	$2q_2 + r_2 = q_1$
$q_1 \bmod 2$	r_2	$r_2 = 1$
		$bucket = n_{dst}/8$

Thus, in the circuit we only need to check that $\exists i : e = P_i \wedge i = n_{dst} \bmod 8$ holds, which is integrated as a linear search over the elements of P and made efficient by favoring the discounted price of math operations in the native field $\mathbb{F}$ over a conditional statement, i.e., we program the lookup as: $\prod_{i=0}^{7}((n_{dst} \bmod 8) - i) + (e - P_i)$, which becomes zero if e is as expected for n_{dst}. Further, to account for the modulo 2 in (4.3), we make the prover supply the answer to the integer division $q_2 = \lfloor q_1/2 \rfloor$ and corresponding remainder $r_2 = q_1 \bmod 2$, which we verify inside the circuit by checking that $2q_2 + r_2 = q_1$. Note here that r_2 corresponds to the left-hand side computation of the latter equation in (4.3). Thus, we know to accept n_{dst} as a neighbor if all equations hold and $r_2 = 1$.

In total, the circuit accepts five additional secret inputs, e, q_1, r_1, q_2, r_2, which work as hints for the circuit to more efficiently check whether a particular destination node n_{dst} is legal by verifying a system of linear equations. Similar to the adjacency list $\mathcal{AL}'$, which contains n_{cur}'s $(bucket, rems)$ pairs pre-split for every transition as we walk over the execution path, the circuit accepts the additional five secret inputs that prove a transition's destination node's validity as a two-dimensional (sequentially accessed) array, π_{exists}, which is also ordered by transitions. See Table 4.1 for a summary of the input variables (hints), how they are computed, and how they are verified inside the circuit.

4.6.2 Final Design of the ZEKRA Circuit

Fig. 4.4 shows the high-level code with the optimizations described in Section 4.6.1, which is compiled into the ZEKRA circuit. As secret input, the circuit accepts: an encoded adjacency list $\mathcal{AL}$ of length N (or padded to equal N) representing the attested program's CFG, some execution path $\mathcal{EP}$ of length E (or padded to equal E) corresponding to the recorded execution path, the mapping $\mathcal{M}$ for translating the attested program's addresses into numeric labels, the random padding used as a blinding factor for the adjacency list (r_1), execution path (r_2), and mapping (r_3), respectively, the auxiliary adjacency list $\mathcal{AL}'$ containing the decoded $bucket, rems$ pairs of the expected $\mathcal{AL}$ entry for each transition, the translation of the execution path addresses into numeric labels $\mathcal{L}$, and finally, the auxiliary proofs π_{exists} to help verify each transition's destination node's validity. As public input, the circuit accepts: the digests of the adjacency list (h_1), execution path (h_2), and the mapping (h_3), respectively, an initial node $n_{\triangleright}$, a final node $n_{\blacktriangleleft}$, and the verifier's nonce nce. Note that to transform our high-level program into a circuit, we must clearly define the bounds of all data structures we want to be expressed (i.e., for which we

want constraints to be generated). However, while we can easily pad adjacency lists to match N before supplying them as input (thus supporting different program complexities with the same circuit), the same does not immediately apply to the execution paths. Note that the execution path through a program might vary drastically between different executions. Thus, to support varying sizes of execution paths, it must be padded to the appropriate length (E) by appending "empty" transitions, which is detected in the circuit when $jmpkind = 2$. Furthermore, to allow the translation of the padded transitions through $\mathcal{M}$, we must reserve an entry ($\mathcal{M}$+1) whose value is used as both the destination and return addresses in each padded transition.

Given the inputs, we verify that the secret adjacency list, attested execution path, and mapping are indeed the correct preimages of the corresponding public digests by first compressing and then hashing each data structure using our implementation of the circuit-friendly Poseidon [82] hashing function H as described in Section 4.5.4. Assuming that the digests were correct, the circuit proceeds to verify that the worker's translation $\mathcal{L}$ of the execution path addresses into their corresponding numeric labels was done correctly according to $\mathcal{M}$. Then, knowing that $\mathcal{L}$ accurately reflects the attested execution path in the abstract world of $\mathcal{AL}$, the circuit proceeds to traverse $\mathcal{AL}$ using $\mathcal{L}$ to verify that the execution path: (i) began at the expected $n_{\triangleright}$, (ii) contains only transitions that are conformant to the adjacency list, and (iii) ends at the expected final node $n_{\blacktriangleleft}$. The traversal is done by instantiating a state variable n_{cur} to $n_{\triangleright}$, which, as we iterate over each transition $(jmpkind, n_{dst}, n_{ret})$ in the execution path, we verify that n_{dst} is a valid neighbor of n_{cur} by consulting n_{cur}'s auxiliary adjacency list, as follows. We first retrieve n_{cur}'s encoded neighbors by accessing n_{cur}'s encoded entry in $\mathcal{AL}$ (which is expensive since RAM is expensive), which we compare against the current transition's n_{dst}'s entry in $\mathcal{AL}'$ (whose access is cheap since we access it sequentially). If the entries match, we know that we can securely rely on $\mathcal{AL}'$ to accurately report n_{cur}'s neighbors, which we leverage to efficiently verify whether n_{dst} is a valid neighbor of n_{cur} by verifying that the current transition's proof in π_{exists} is valid with respect to some $bucket, rems$ pair in $\mathcal{AL}'$ as described in Section 4.6.1.3. If n_{dst} is determined to be a valid neighbor, we update n_{cur} to n_{dst} and progress to the next transition. Moreover, while iterating over the execution path, we maintain a shadow stack to ensure exact back edge integrity similar to other CFA schemes [176, 169]. Finally, once we have walked the full path, we verify that n_{cur} reached $n_{\blacktriangleleft}$. The circuit is only satisfied if *all* verifications were successful.

4.7 Empirical Performance Evaluation

Our evaluation addresses the questions of: (i) how efficient is ZEKRA for different program complexities and (ii) how tolerable are the combined costs for CFA of real-world *deeply embedded applications*.

4.7.1 Asymptotic Performance

Table 4.2 shows the complexity of our design considering a 254-bit field $\mathbb{F}$ and a Poseidon-128 implementation with an arity/rate of 8 field elements and a cost of $\mathfrak{C} = 405$ constraints per call. For comparison, we also show the complexity *without* the space efficient adjacency list encoding described in Section 4.5.4, i.e., where each entry in $\mathcal{AL}$

```
1 : // public and secret circuit inputs
2 : public{h_1, h_2, h_3, n_▷, n_◀, nce}
3 : secret{AL[N], EP[E][3], M[N+1], r_1, r_2, r_3,
4 :        AL'[E][2ℓ], L[E][2], π_exists[E]} // hints
5 : external{ // code executed by worker to compute hints
6 :   n_cur ← n_▷ // keep state during traversal
7 :   for i = 0...E do // compute hints for each step
8 :     AL'(i) ← split(AL(n_cur))
9 :     L(i)(dst) ← {j|M(j) = EP(i)(dst)}
10 :    L(i)(ret) ← {j|M(j) = EP(i)(ret)}
11 :    π_exists ← Table 4.1(EP(i)(dst), AL'(n_cur))
12 :    n_cur ← L(i)(dst)}
13 : // circuit code
14 : shadowStack[D]
15 : Vf(H(AL‖r_1) = h_1)
16 : Vf(H(EP‖nce‖r_2) = h_2)
17 : Vf(H(M‖r_3) = h_3)
18 : Vf(∀i ∈ {0...E} : M(L(i)(dst)) = EP(i)(dst)∧
19 :                   M(L(i)(ret)) = EP(i)(ret))
20 : n_cur ← n_▷ // keep state during traversal
21 : for i = 0...E do // walk the execution path
22 :   jmpkind ← EP(i)(jmpkind)
23 :   (n_dst, n_ret) ← (L(i)(dst), L(i)(ret))
24 :   if jmpkind ≠ ∅ then // not an empty (padded) transition
25 :     bucket ← ⌊n_dst/8⌋
26 :     pos ← n_dst mod 8
27 :     e, q_1, r_1, q_2, r_2 ← π_exists(i)
28 :     Vf((∑_{j=0,2,...}^{2ℓ-1} AL'(i)(j+1)2^{⌊j/2⌋(bucket bitwidth+8)+8
29 :           + AL'(i)(j)2^{⌊j/2⌋(bucket bitwidth+8)}) = AL(n_cur))
30 :     Vf((∏_{j=0}^{7}(pos − j) + (e − P_j)) = 0)
31 :     Vf(2q_2 + r_2 = q_1)
32 :     Vf(r_2 = 1)
33 :     Vf(∃j ∈ {0, 2, ..., 2ℓ − 2} :
34 :         AL'(i)(j) = bucket∧
35 :         AL'(i)(j+1) = e · q_1 + r_1)
36 :     n_cur ← n_dst // progress CFG state
37 :   if jmpkind = call then
38 :     push(n_ret, shadowStack)
39 :   elseif jmpkind = ret then
40 :     Vf(pop(shadowStack) = n_dst)
41 : Vf(n_cur = n_◀)
```

Figure 4.4: The high-level ZEKRA program code, which can be compiled into an outsourceable circuit.

is simply the concatenation of that node's neighbors, where Δ denotes the maximum supported neighbors of any node.

The first four rows in Table 4.2 give the complexity of verifying: the adjacency list (with and without encoding), attested execution path, and mapping, respectively. Note

Table 4.2: Component complexity in terms of the number of constraints when compiled using xJsnark, where $\mathfrak{C} = 405$ is the cost per call to Poseidon. The table also shows the cost if we store digests in $\mathcal{AL}$ to emulate more space (beyond $\mathbb{F}$'s bitwidth).

Circuit Component/Gadget	Complexity	Actual (Asymptotic) Total Cost	Using Poseidon digests as $\mathcal{AL}$ elements
C1: $\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{AL}}\|\overline{r_1}) = h_1)$	$O(N)$	$\mathfrak{C}\lceil(N\ell(bucket\text{'s bitwidth} + 8) + 1)/(8\mathbb{F}\text{'s bitwidth})\rceil$	$\mathfrak{C}\lceil N/8\rceil$
- (wo. $\mathcal{AL}$ encoding)	$O(N)$	$\mathfrak{C}\lceil(N\Delta label\text{'s bitwidth} + 1)/(8\mathbb{F}\text{'s bitwidth})\rceil$	$\mathfrak{C}\lceil N/8\rceil$
C2: $\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{EP}}\|\overline{nce}\|\overline{r_2}) = h_2)$	$O(E)$	$\mathfrak{C}\lceil(E2addr\text{'s bitwidth} + 4)/(8\mathbb{F}\text{'s bitwidth})\rceil$	N/A
C3: $\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{M}}\|\overline{r_3}) = h_3)$	$O(N)$	$\mathfrak{C}\lceil(Naddr\text{'s bitwidth} + 1)/(8\mathbb{F}\text{'s bitwidth})\rceil$	N/A
C4: $\mathsf{Vf}(\overline{\mathcal{M}}(\overline{\mathcal{EP}}) = \overline{\mathcal{L}})$	$O(E\sqrt{N})$	$2E(2\sqrt{N} + \log_2 N) + 10E$	N/A
C5: Forward edge integrity	$O(E\sqrt{N})$	$E(2\sqrt{N} + \log_2 N) + E(\ell + 38 + label\text{'s bitwidth})$	$+E\mathfrak{C}\lceil(\ell(bucket\text{'s bitwidth} + 8))/(8\mathbb{F}\text{'s bitwidth})\rceil$
- (wo. $\mathcal{AL}$ encoding)	$O(E\sqrt{N})$	$E(2\sqrt{N} + \log_2 N) + E(\Delta + 11 + label\text{'s bitwidth})$	$+E\mathfrak{C}\lceil(\Delta label\text{'s bitwidth})/(8\mathbb{F}\text{'s bitwidth})\rceil$
C6: Backward edge integrity	$O(E\sqrt{D})$	$2E(2\sqrt{D} + \log_2 D) + E(28 + 2\log_2 D)$	N/A

that it takes a single constraint to verify that a computed digest matches its corresponding public reference. Thus, we only consider the complexity of the hashing. Note here that the bit-space needed when compressing the unencoded adjacency list into the least number of field elements (to minimize the number of calls to Poseidon) is directly affected by the number of supported neighbors Δ (second row). In contrast, the bit-space needed for the encoded adjacency list is affected by the number of levels (ℓ) used for the encoding (first row). To illustrate the power of the encoding, note that we can only store 25 10-bit labels in a 254-bit field element. Thus, since we only have a one-dimensional adjacency list of field elements, we can only represent adjacency lists with $\Delta = 25$. However, using the encoding, we can store a total of 16 levels ℓ of 15-bit ($bucket, rems$) pairs (we need 7-bit buckets when considering 10-bit labels), which can hold 128 labels. Thus, using the encoding, we can reduce the number of calls to Poseidon and also represent more complex adjacency lists using fewer bits.

Note that verifying the correctness of the worker's translation of an execution path of length E requires $2E$ accesses to $\mathcal{M}$ inside the circuit since we must verify each transition's destination address and (possible) return address. This complexity is shown in row five of Table 4.2, which evidently dominates the overall circuit complexity.

The complexity of verifying that each transition's destination node is valid according to an encoded or unencoded adjacency list is shown on rows six and seven of Table 4.2, respectively. Note that we accept the pre-split version of $\mathcal{AL}$ as a sequentially accessed, two-dimensional structure $\mathcal{AL}'$ in both cases. For the encoded version, $\mathcal{AL}'$ contains the ($bucket, rems$) pairs as described in Section 4.6.1. For the unencoded version, $\mathcal{AL}'$ contains the individual neighbors. In both cases, we maintain our state variable (n_{cur} in Fig. 4.3) as an unsigned integer, which is used to access $\mathcal{AL}$ and whose bitwidth is determined by the maximum label. Furthermore, in both cases, we perform a linear search over $\mathcal{AL}'$ to find a match, which requires either ℓ (using a step size of 2) or Δ iterations with and without the encoding. (Note that Δ quickly outgrows ℓ.) Finally, while negligible, note that the slightly higher (constant) cost per transition in the case of the encoded version is the cost of our proposed method of verifying a linear system of equations as described in Section 4.6.1.3 to check if a destination node exists in a ($bucket$, $rems$) pair.

Recall that a verifiable program's complexity in terms of the number of constraints it compiles down to is the sum of the cost of all branches as described in Section 4.6.1.

Thus, the complexity of the back-edge integrity component (last row) includes the sum of both branches (push and pop) per transition. However, note that we can usually keep the stack depth, D, small (unless attesting to highly nested/recursive code). Hence the double memory cost for this component is less significant than that of the fourth component.

4.7.1.1 Supporting More Neighbors

To store more neighbors in $\mathcal{AL}$, we need more space per node element. Without compiler support for two-dimensional RAM, a naive approach is to emulate it with more arrays. However, in this case, the memory access cost grows proportionally to the number of arrays, i.e., for two parallel arrays, the cost per transition becomes $2 \times (2\sqrt{N} + \log_2 N)$. Another approach, whose cost is also shown in Table 4.2, is to instead store the hash of a node's neighbors as a field element in $\mathcal{AL}$ whose preimage is then given as a two-dimensional array in $\mathcal{AL}'$. Then, for each transition i we simply verify that $\mathsf{H}(\mathcal{AL}'(i)) = \mathcal{AL}(n_{cur})$ before performing a neighbor lookup in $\mathcal{AL}'(i)$, where $\mathcal{AL}'(i)$ now supports an arbitrarily large neighbor space. Note, however, that this choice comes at a cost proportional to the number of Poseidon calls we need to make per transition to perform the verification and thus is only mentioned as an alternative method for our approach to scale in support of attesting to arbitrary CFG complexities. Specifically, to get 8 field elements (the considered arity) of neighbor storage per node (allowing for ≈ 1024 neighbors using the encoding when considering 10-bit labels), this comes at the cost of one Poseidon call per transition, i.e., giving an overall (additional) cost of $E \times \mathfrak{C}$.

4.7.2 Empirical Performance

4.7.2.1 Datasets

Table 4.3 shows some extracted datasets for a selection of demonstrative applications taken from the embench-iot suite [68], which comprises a set of real-world, deeply embedded applications[2]. To ensure reproducibility, we coded helpers [56] to perform all evaluation steps. For compilation, we use GCC options `-Os -g0` and the `-fno-optimize-sibling-calls` flag for deactivating *sibling and tail recursive calls* optimizations. We then use the angr [164] binary analysis tool for extracting static CFGs and sample execution paths through symbolic execution, where the sample paths simulate paths as recorded by a prover. To generate the trusted reference material, we use the NetworkX Python package [87] for translating the extracted CFGs into isomorphic, numerically labeled representations, which are then converted into corresponding adjacency lists $\mathcal{AL}$ and used to derive the address-to-label mappings $\mathcal{M}$.

4.7.2.2 Labeling

Note that the minimum number of levels (ℓ) needed to encode a specific adjacency list is determined by the maximum number of quotients (i.e., buckets) shared by any node's neighbors, which depends on the way the nodes are labeled numerically. In our experiments, we labeled each extracted CFG's nodes using consecutive integers in the order

[2] Note that these applications only served as data points in our performance evaluation and were not selected by their need for control-flow attestation in practice.

Table 4.3: Sample datasets from the embench-iot suite of real-world embedded applications, each with 24-bit address space.

	Control-Flow Graph G				Sample recorded execution path (through symbolic execution)					
Application	N	**Edges**	$\Delta(G)$	ℓ	E (pre[a])	E (post[b])	**# loops**	**Avg. loop length**	**Avg. # of repetitions**	D
aha-mont64	114	151	6	2	997	110	109	2.82 ($\sigma \approx 1.55$)	3.01 ($\sigma \approx 1.50$)	5
crc32	88	106	2	2	3,090	24	1	3.00 ($\sigma \approx 0.00$)	1,023 ($\sigma \approx 0.00$)	6
cubic	147	198	8	3	105	10	1	1.00 ($\sigma \approx 0.00$)	96.00 ($\sigma \approx 0.00$)	5
edn	152	195	2	2	4,889	422	16	5.81 ($\sigma \approx 12.34$)	89.19 ($\sigma \approx 98.89$)	5
huffbench	188	284	3	3	9,894	1,155	84	4.39 ($\sigma \approx 4.27$)	53.77 ($\sigma \approx 155.81$)	6
matmult-int	113	143	8	2	37	37	0	0.00 ($\sigma \approx 0.00$)	0.00 ($\sigma \approx 0.00$)	5
md5sum	129	176	4	3	8,399	382	6	54.33 ($\sigma \approx 119.26$)	537.17 ($\sigma \approx 490.16$)	7
minver	176	252	3	3	324	201	21	5.00 ($\sigma \approx 6.20$)	3.10 ($\sigma \approx 2.43$)	5
nbody	113	140	3	2	108	58	4	5.00 ($\sigma \approx 0.00$)	3.50 ($\sigma \approx 1.12$)	6
nettle-aes	156	211	3	3	1,821	199	11	14.00 ($\sigma \approx 15.21$)	34.27 ($\sigma \approx 71.00$)	7
nettle-sha256	173	245	4	3	295	106	10	2.70 ($\sigma \approx 3.16$)	17.60 ($\sigma \approx 19.64$)	6
nsichneu	853	1500	2	2	659	655	1	4.00 ($\sigma \approx 0.00$)	2.00 ($\sigma \approx 0.00$)	4
picojpeg	633	1168	37	7	1,875	335	6	12.50 ($\sigma \approx 8.46$)	31.67 ($\sigma \approx 20.90$)	11
primecount	105	127	3	3	1,742	1,001	66	8.62 ($\sigma \approx 11.69$)	2.36 ($\sigma \approx 0.64$)	5
sglib-combined	728	1084	5	3	8,191	868	105	12.63 ($\sigma \approx 16.62$)	7.90 ($\sigma \approx 20.19$)	8
slre	347	528	6	2	2,217	609	5	13.40 ($\sigma \approx 12.47$)	10.80 ($\sigma \approx 16.12$)	13
st	123	155	3	3	1,143	65	7	1.57 ($\sigma \approx 0.90$)	99.00 ($\sigma \approx 0.00$)	6
statemate	434	657	2	2	256	102	3	1.34 ($\sigma \approx 0.47$)	31.67 ($\sigma \approx 22.16$)	6
tarfind	102	137	3	3	12,070	443	38	147.0 ($\sigma \approx 161.36$)	13.82 ($\sigma \approx 40.87$)	5
ud	133	174	3	2	393	185	25	3.76 ($\sigma \approx 2.20$)	4.60 ($\sigma \approx 7.31$)	5
wikisort	425	645	7	3	4,610	152	7	26.71 ($\sigma \approx 50.41$)	66.57 ($\sigma \approx 84.31$)	7

[a]before and [b]after path compression (removing all consecutively repeated sequences).

they appeared. (We defer the graph optimization problem of finding the most optimal ordering as future work.) Note, however, that even without any special preprocessing, we can already observe in Table 4.3 for `picojpeg` how "consecutivity" among the neighbors of a node allows us to effectively represent the maximum outdegree $\Delta = 37$ using only $\ell = 7$ levels, meaning we need only 105 bits to encode each node's neighbors (each level is 15 bits). Without the encoding, we would need 370 bits to represent 37 neighbors, which already exceeds the considered prime field $\mathbb{F}$.

4.7.2.3 Compression

As noted in [2] and utilized in most CFA works, the most basic method of reducing the path explosion problem without loss of accuracy is to prune repetitions in the execution path since they do not affect the *legality* of the *control-flow*. Similarly, we consider that recorded execution paths are compressed such that each unique loop path only occurs once, i.e., all consecutively repeating sequences are discarded. Note that this compression *only removes duplicates* in the path, allowing us to use a smaller circuit for verification. However, the compression does *not* affect our ability to detect control-flow attacks (except those that only affect the number of loop iterations). (We discuss extending our approach to attesting to the number of loop iterations in Section 4.8.)

Table 4.4: This table shows the average time (and standard deviation, σ), after 10 iterations, for the worker to generate proofs over ZEKRA circuits compiled to support different attestation data sizes. Proof verification takes $\approx$ 2 ms in all cases.

Circuit Config (Data Structure Sizes)							**Compiled Circuit w. Component Workload Dist. (in %)**							**Worker**
E	N	D	ℓ	*label*	*bucket*	*addr*	pk_C **(MB)**	**# Const.**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	Prove **(avg. s)**
500	500	15	15	10 bits	7 bits	24 bits	64.638	336,230	7.6	1.6	0.8	38.3	32.1	19.6	4.316 ($\sigma \approx 0.007$)
500	500	15	15	10 bits	7 bits	88 bits	69.327	366,605	7.0	7.0	3.5	35.1	29.5	18.0	4.709 ($\sigma \approx 0.001$)
1000	1000	15	15	10 bits	7 bits	24 bits	134.180	703,669	7.3	1.5	0.7	39.3	32.5	18.7	8.974 ($\sigma \approx 0.057$)
1000	1000	15	15	10 bits	7 bits	88 bits	143.888	764,419	6.7	6.7	3.3	36.1	29.9	17.3	9.730 ($\sigma \approx 0.008$)
1200	1000	15	15	10 bits	7 bits	24 bits	158.907	809,043	6.3	1.6	0.7	39.4	32.5	19.6	10.552 ($\sigma \approx 0.049$)
1200	1000	15	15	10 bits	7 bits	88 bits	170.306	877,893	5.8	7.0	2.9	36.3	30.0	18.0	11.374 ($\sigma \approx 0.026$)

4.7.2.4 Experimental Setup

As described in Section 4.6.1, we implemented our solution using xJsnark [114], a high-level code-to-circuit compilation framework that employs a mix of optimizations to minimize circuit complexity. The high-level code is compiled into arithmetic circuits in an extension of the Pinocchio [147] intermediate opcode format, which, using the jsnark interface [113], are translated into the R1CS constraint system and fed into the libsnark [119] backend for instantiating a particular zkSNARK proof system over the circuit. In our case, we considered libsnark's implementation of the state-of-the-art Groth16 [84] proof system (over the BN128 curve), whose proof is 1016 bits and contains 3 group elements (2 $\mathbb{G}_1$ elements and 1 $\mathbb{G}_2$ element), and 3 pairings dominate verification. We then benchmarked our prototype using libsnark's built-in profiler, which includes the generation of the circuit's proving and verification keys and execution of the proof generation and verification algorithms on our experimental inputs, which were formatted from our real-world extracted datasets. As the worker, we considered a machine with an AMD Ryzen 7 3700X processor and 16 GB of memory (experiments were conducted in a WSL2 environment).

4.7.2.5 Benchmarks

Table 4.4 shows benchmarks for generating proofs over some demonstrative circuits (proof verification is constant due to Groth16's underpinnings). Note that when compiling a particular circuit, we must define the sizes of the data structures we want to express. However, even if we fix both E and N to 1K, we can still supply execution paths and adjacency lists $\leq$ 1K by employing padding as described in Section 4.6.2. Thus, the larger circuits in Table 4.4 (after the fourth row) support all datasets shown in Table 4.3. Furthermore, note that the reported running times assume that the proving key is preloaded in memory, which holds when proof generation is performed by dedicated workers that expect the key to be used regularly and thus retain it in memory. Further, note that we only list the proving key sizes since Groth16's verification keys have a constant size of 3312 bits. From the reported timings, it is evident that we prove the satisfaction of arithmetic circuits at a rate of $\approx$ 77.8 constraints/ms on our considered experimental setup.

Whereas the complexity of VC methods that fully convert programs into circuits to verify each emulated CPU cycle's correctness grows with a program's control-flow and

assembly complexity as described in Section 4.2, our coarse-grained approach grows only to the control-flow complexity. For example, the cost per CFG edge (i.e., control-flow) is ≈ 674 constraints for the fifth circuit in Table 4.4, which is less than the per-cycle cost of $\approx 1,458$ in [20]. (Note also that there are inherently more CPU cycles than control-flow transitions during a program's execution.) Thus, while we require a trust anchor on the prover to record and authenticate a program's execution path, we scale better to larger programs. Finally, contrary to circuits crafted for particular programs, our circuit allows attesting to the execution of arbitrary programs via its inputs.

4.7.2.6 *Attesting to Executed Instructions*

While generally not considered for CFA since the attested program is generally considered to reside in DEP-protected memory, [186] proposed having the prover hash the executed instructions along with the BBL addresses to detect TOCTOU attacks when considering physical (non-invasive) adversaries who can manipulate program code during runtime. In this case, the only change in the recorded execution path $\mathcal{EP}$ is that it contains digests instead of the destination and return addresses, i.e., it becomes a sequence of transitions of the form $(jmpkind, d_{dst}, d_{ret})$, where $d = \mathsf{H}(addr \| instructions)$. We can easily support this approach by initially storing $\mathsf{H}(addr \| BBL_{addr})$ as the elements of the translator $\mathcal{M}$ instead of only storing the addresses. Further, note that only the hashing of the attested execution path $\mathcal{EP}$ and the translator $\mathcal{M}$ are affected (rows 3 and 4 of Table 4.2). Specifically, assuming 88-bit digests as in [186], each transition will occupy 178 bits (destination and return address and 2 bits for the jumpkind).

4.7.2.7 *Benchmarks for Other Proof Systems*

The jsnark interface alternatively supports libsnark's implementation of the optimized Pinocchio zkSNARK proof system [147] as proposed in [20]. However, using this latter system [20] over Groth16 [84] showed an increase of $\approx 13\%$ in proving time and $\approx 45\%$ in proving key size for the largest circuit in Table 4.4, including a larger proof size of 2294 bits (7 $\mathbb{G}_1$ elements and 1 $\mathbb{G}_2$ element). While current SNARK technology is on the borderline of feasibility, proof systems are evolving increasingly, and thus we expect to handle (and optimize) larger arithmetic circuits more efficiently shortly. Furthermore, we note that the workload requirements of the worker can be further scaled up using current systems like DIZK [182], which allows the generation of proofs to be distributed across machines (workers) in a compute cluster (e.g., EC2). Additionally, note that proof systems have also recently emerged that outperform the Groth16 proof system, such as SpartanSNARK [160], which, compared to libsnark's implementation of Groth16, appears to be 2× faster on the worker. See the corresponding works for details on their proving time and key size complexities.

4.8 Discussion and Security Properties

4.8.1 Rejection of Control-Flow Attacks

To evaluate ZEKRA in terms of detecting control-flow attacks (i.e., preventing proofs from being accepted when execution paths are illegal), we tested several paths bearing

real code injection or ROP/JOP attack patterns. Note that the ZEKRA circuit, when compiled, is a concise set of constraints (or equations) that is satisfied only when *all variables hold all equations* simultaneously. This constraint system includes both the set of constraints from the forward-edge integrity component, which requires jumps and calls to target *valid neighbors*, and the set of constraints from the backward-edge component requiring returns to target the *contextually-valid* node. Thus, if an execution path contains *any* transition that causes *any* constraint to fail, then the verifier rejects the proof. Thus, as expected, conventional control-flow hijacking attacks that employ code injection result in rejected proofs since they add transitions to the execution path which target nonexistent CFG nodes.

Similarly, code-reuse attacks such as ROP and JOP are detected and rejected by the verifier since they cause an execution path to contain transitions that target invalid neighbors. Furthermore, if a function's return address is hijacked to execute a malicious gadget sequence, then this will also cause an unfulfilled constraint in the backward-edge integrity component since the shadow stack's topmost entry will not match this new node. Furthermore, note that the execution path must also, at a minimum, be translatable to its numeric representation according to the program-specific address-to-label mapper $\mathcal{M}$. Therefore, in the case of control-flow hijacking, e.g., when stitching together a chain of gadgets for ROP, where an adversary might include branches to unexpected offsets within a BBL instead of its starting address, this is always caught since $\mathcal{M}$ will not include such entries. However, note that the circuit is limited to detecting control-flow attacks, i.e., DOP attacks (see Fig. 4.1), even impure, will remain undetected unless the reference CFG forces legal executions through designated routes in the CFG.

4.8.2 Comparison with CFA Works

While our scheme suffers on the intermediate worker due to the computational resources currently needed to generate zkSNARK proofs[3], we achieve (i) optimal cost on verifiers, (ii) optimal transmission overhead (towards the verifier), and (iii) stronger security properties, as opposed to all existing CFA works [2, 63, 186, 62, 3, 176, 95, 125, 169, 117, 140, 187]. Both (i) and (ii) are due to the "succinctness" of zkSNARK proof constructions, where verification is unaffected by circuit complexity, and the proof size is constant, e.g., with Groth16, the proof only contains three group elements (totaling 1019 bits). Moreover, besides the proof, the verifier only needs to receive the circuit's public inputs, which all have constant sizes, comprising the considered entry and exit node labels, three digests, and the verifier's nonce (whose echo signifies freshness). Note that in other CFA schemes, the prover generally transmits the full execution path and a corresponding digest directly to the verifier. Furthermore, with all other CFA schemes, all verifiers are assumed to maintain an extensive reference database of all the possible execution paths [2] (or, more commonly, the attested program's CFG [62] due to the difficulty of exhaustively discovering all such paths beforehand as described in Section 4.3.3) to check the legality of attested execution paths. We cover (iii) in Section 4.8.5.

[3]Which makes it challenging to target complex software, such as that considered by ScaRR [176] and ReCFA [187], using our scheme due to the worker resources needed.

4.8.3 Execution Path Compression

While not related to our protocol's effectiveness, the considered granularity of the program CFG and execution paths directly affects the efficiency and scalability of our approach. As also noted by other works [2, 63, 186], we can, without sacrificing accuracy, decrease granularity to increase code coverage by pruning unnecessary edges in a CFG and ignoring repetitions in the execution path. For example, to simplify a CFG, we can, similar to inlining, where callee functions are inlined into the caller functions to reduce complexity, fuse connected nodes where only one path exists between the nodes, e.g., the nodes n_2, n_3 and n_7 in Fig. 4.1. This, however, requires that the prover's trust anchor can identify and translate execution paths into their succinct form, e.g., if it observes the path $n_1 \rightsquigarrow n_3 \rightsquigarrow n_8 \rightsquigarrow n_4$ during program execution, then it records it as a function call from n_1 to n_8 and a function return to n_4. Similarly, to mitigate path explosion caused by loops, the prover can notice when a sub-path begins consecutively repeating itself, e.g., $n_5 \rightsquigarrow n_6 \rightsquigarrow n_5 \rightsquigarrow n_6$, and record it as only occurring once as considered in our evaluation in Section 4.7.2.

4.8.4 Current Limitations

However, on a contrary note, while the current ZEKRA circuit puts no restraints on the number of loop iterations, ensuring a correct (or safe) number of loop iterations can be significant depending on the application [2]. Thus, we note the possibility of extending the circuit to accept a secret policy specifying such loop (upper and lower) bounds. Furthermore,- while we currently assume a *semi*-untrusted worker, future work can weaken this trust assumption as described in Section 4.4.4.1. Finally, besides further optimizations to reduce the computational resources needed by workers, future work should investigate the viability of constructing a similar scheme with post-quantum-secure proof systems, e.g., using STARKs [18].

4.8.5 Security Properties

Besides its secure implementation, our scheme's foundational security is given by the underlying proof system's security (and circuit compiler). Note that for all configurations we ran our prototype on (hundreds of executions), we recorded no completeness errors. In the following, we give an intuitive description of the different security properties our scheme is designed to provide and how it achieves them.

Property 4.8.1 (Proof Unforgeability). Not only does our scheme have negligible error probability for falsely rejecting or falsely accepting an execution path or claims about an execution path as authentic, but our scheme is also provably secure against (computationally bounded) adversaries attempting to pass manipulated execution paths as authentic due to the utilization of an accompanying signature scheme with the considered zkSNARK proof system. Specifically, the verifier detects any adversary (e.g., the worker) attempting to pass in an unauthentic execution path as part of the proof. For example, suppose that (i) the execution path, the nonce, or the random noise passed in secret to the proof generation result in a different digest than the public digest, or (ii) the adversary computes and inputs benign values to the proof generation but the public digest differs from the prover's signed digest. In the first case (i), the verifier rejects the

proof in the proof verification stage due to unmet constraints, and in the latter case (ii), the verifier rejects the proof in the signature verification stage due to a digest mismatch. Therefore, the worker cannot convince the verifier that an illegal (unsigned) execution path (e.g., one that reveals the presence of an attack against the attested program on the prover) is legal.

Property 4.8.2 (Zero Knowledge). Our scheme ensures that a verifier only learns whether the expected prover freshly recorded an execution path and whether the execution path is legal according to the secret adjacency list's ($\mathcal{AL}$) preimage of a specific trusted reference digest. Specifically, we ensure that no execution path or CFG details are disclosed to the verifier by accepting them as secret circuit inputs and making the reference digests statistically-hiding commitments.

Property 4.8.3 (Forward & Back Edge Integrity). The ZEKRA circuit is only satisfied if the provided execution path is legal according to the given $\mathcal{AL}$. The rules are simple. Transitions target adjacent nodes, and back transitions are shadow stack compliant. By verifying that the execution path consistently flows through adjacent nodes in the forward direction (using the adjacency list) and that callees always return to the contextually correct caller (using the shadow stack), we ensure fine-grained detection of all control-based attacks as described in Section 4.8.1.

Property 4.8.4 (Execution Path Completeness). Like how a CFG precisely models all program executions, we support attesting to any execution path in a program's CFG.

4.9 Conclusions

We presented ZEKRA, a novel and effective protocol that utilizes the combined strength of verifiable computation and control-flow attestation to enable underpowered provers to convince untrusted verifiers about the correct control-flow execution of deeply embedded programs in zero knowledge. The proposed scheme guarantees the attested program's forward and back-edge correctness according to its reference CFG, using several optimizations for representing and traversing CFGs. While currently only demonstrated for deeply embedded applications, our research suggests verifiable computation based on zkSNARK constructions as a feasible direction for enhancing CFA schemes with additional privacy guarantees.

Chapter 5

RETRACT: Expressive Designated Verifier Anonymous Credentials

Abstract

Anonymous credentials (ACs) are digital cryptographically-secure versions of paper and digital credentials that let us selectively prove possession of encoded attributes (claims) to verifiers such as digital services, employers, or government departments without disclosing any other information. While attributes by governmental issuers usually reflect basic personal information about the credential holder (e.g., name, gender, age, address), attributes can also reflect more extensive claims about holders, such as the holder's platform details and configuration. Since the attributes might be sensitive, it is popular to embed additional attributes in the credential about the existing attributes, e.g., that age is above 18, thus allowing a holder to show that their age satisfies some condition without revealing the exact age. However, since each verifier might have different policies that must be satisfied, it is becoming increasingly impractical for issuers to embed all possible claims in a credential. To mitigate this problem and allow arbitrary policies to be checked against individual attributes without complicating or overwhelming the credential, we propose to let verifiers dynamically define policies as high-level programs which can be *verifiably* executed by holders on their credentials. Furthermore, to mitigate the potential risk of dishonest verifiers attempting to benefit or otherwise leak sensitive information learned through this unlimited expressiveness of policies, we propose making the proofs *designated verifier*. Thus, any proof produced for one verifier cannot be used to convince another.

5.1 Introduction

An anonymous credential system allows an entity to obtain a verifiable credential from an issuer containing several attributes, or claims, which it can later prove possession of to a

verifier. However, instead of revealing the entire credential to the verifying party, anonymous credentials allow the holder to selectively disclose only a subset of the attributes. Such digital credentials are becoming increasingly popular as they provide a supposedly secure and meaningful way for entities to bear cryptographic evidence reflecting their properties and claims. For example, to access cloud-based services, entities can directly prove that they bear the necessary attributes to satisfy the service's access policy without involving their identity providers. One of the most well-known cryptographic methods that provide the necessary properties to build such anonymous credentials is BBS+ signatures [11], which have become widely adopted as a standard for building anonymous credentials.

Besides selective disclosure, it is becoming increasingly popular to incorporate zero-knowledge proofs [178, 65, 33, 152, 42, 66, 179, 96, 170], where credential holders can prove that they possess attributes that satisfy some condition without revealing any other information. However, several schemes still only support a limited range of predicates [178, 65, 33, 179, 170], such as basic boolean operators, or being limited to range proofs or set membership proofs. Furthermore, another challenge, as mentioned in [90], occurs when considering anonymous credentials on devices, such as smartphones, where we encounter issues such as credential sharing and the need to cope with fewer computational resources. To address this issue, authors of [90] proposed the notion of core/helper anonymous credentials, where credentials are split between a secure core (e.g., a SIM card) and a more resourceful helper (e.g., a smartphone). Here the idea is that the helper cannot use the credential without the core's help. Though, as the authors mention, the core's effort must be minimal and independent of the credential's size due to its limited resources. Finally, most prior literature on anonymous credentials does not consider the designated verifier property [104]. Specifically, in most schemes, there is no attempt to prevent the verifier from leaking whatever information they learned from the verification process to other third parties. The designated verifier property is needed to protect against such malicious activities by dishonest verifiers. Regardless of whether the holder discloses attributes or proves to possess attributes that satisfy a policy or a combination of the two, it should not be possible for verifying parties to misuse or leak information obtained from the verification process.

Motivating example. As a small motivating example, consider a scenario where university graduates get a digitally signed graduation certificate containing all information related to their completed studies. Then, when a graduate wants to apply for a job, they might have to convince a party that it satisfies some policy defined for that particular job, such as having completed some specific courses, having an average grade above some threshold, and living within proximity. Regardless, the hiring department should not be able to convince anyone else of this information to sell or otherwise misuse the information, which is particularly important to align with the recent European General Data Protection Regulation (GDPR).

5.1.1 Constributions

We present a fully *expRessive dEsignaTed veRifier Anonymous CredenTials* (RETRACT) scheme in the core/helper model of [90] that uses BBS+ signatures for building credentials and incorporates state-of-the-art verifiable computation techniques to allow holders to prove arbitrary predicates on their credentials in zero knowledge. In a nutshell, to com-

bine BBS+ signatures with verifiable computation, we consider commit-carrying zero-knowledge Succinct Non-interactive ARguments of Knowledge (cc-zkSNARK) [84, 37] proof constructions that accept predicates expressed as arithmetic circuits. We then show how to use well-established proof statement composition methods [35] for extending a *proof of knowledge* of a valid BBS+ signature with proof that the commitment contains attributes from the BBS+ signature while also ensuring that the overall scheme remains *designated verifier*. While the generation of zkSNARK proofs is generally considered slow, the proofs are short and verified remarkably fast, making zkSNARKs attractive, especially in Distributed Ledger Technology (DLT), e.g., the anonymous cryptocurrency Zcash and Ethereum, and for verifiable credentials. Our proof-of-concept prototype is accessible online [58].

5.2 Related Works

5.2.1 Flexible Credentials

Following the initial work of Chaum [43], there has been a long line of work (e.g., [14, 32, 90, 33]) with successively more efficient and expressive anonymous credentials that have been widely deployed in several real-world applications, such as U-Prove [145], and Idemix [36]. Recently, we have also witnessed a synergy between anonymous credentials and predicate proofs to improve expressiveness. For example, Trinsic [178] uses BBS+ signatures [11] and allows for range-based predicate proofs using basic arithmetic operators. Similarly, the Decentralized Identity Foundation (DIF) allows some algebraic rules and set membership checks as part of their presentation exchange specification [65]. Another recent solution is Dock [66], which uses BBS+ signatures for creating credentials and has recently upgraded from supporting basic predicate proofs, e.g., membership and range proofs, to supporting more arbitrary predicates expressed in Circom [97] and proven correct using a CP-SNARK version of Groth16 [84], called LegoGro16 [37]. In [42], researchers proposed *zero-knowledge credentials* in the decentralized identity (DID) ecosystem, where holders can employ a general-purpose zkSNARK proof system (described in Section 5.3.5) to produce proofs of arbitrary computations over their credentials. Similarly, authors of [152] also consider zkSNARKs and propose a toolkit for creating complex statements and the composition of credentials.

5.2.2 Designated Verifier

While not an anonymous credential scheme, authors of [89] propose a single-sign-on (SSO) protocol that uses BBS+ signatures [11] and adopts a form of *designated verifier*. Specifically, in their SSO scheme, authentication tags can only be validated by the verifier of the service for which they were designated. However, as the authors note, this version of "designated verifier" is slightly different from that defined initially by Jacobsson [104] (described in Section 5.3.6). Specifically, in [104], the idea of a designated verifier is to prevent the verifier from convincing others about a transcript since the verifier could just as well have generated it, whereas in [89] anyone can verify the signer of authentication tags. In [64], authors propose functional credentials based on homomorphic attribute-hiding predicate encryption schemes. The idea for holders to prove statements is: given a

ciphertext encoding a given policy, a holder decrypts the ciphertext to convince a verifier that they know a key for a set of attributes that matches the policy. However, in their scheme [64], the idea is to keep policies secret from designated verifiers. While useful in certain applications, this definition of the designated verifier also differs from that of [104]. Finally, [61] propose using smooth projective hash functions (SPHF) [52] to allow holders to make designated-verifier proofs.

5.2.3 Core/Helper Setting

As mentioned in [90], a prominent example of the core/helper setting is the Direct Anonymous Attestation (DAA) protocol [29], which was designed for privacy-preserving remote attestation of platforms. Here the core device is the Trusted Platform Module (TPM) [172], and the helper is the hosting platform to which the TPM is connected. While DAA is technically a group signature protocol for creating anonymous signatures (with optional linkability) on messages to convince a verifier that an authorized TPM signed a message, some recent works extend the DAA protocol with attributes (DAA-A), and selective disclosure [46, 31, 30]. While [46] considered CL [34] and SDH [25] signatures for credentials, the recent DAA-A schemes [31, 30] shifted to q-SDH BBS+ signatures [11]. However, noting how DAA-A schemes are tailored towards a specific core (i.e., the TPM), authors of [90] proposed core/helper anonymous credentials (CHAC), which uses a combination of signatures with flexible public keys (SFPK) [12] and a novel notion of aggregatable attribute-based equivalence class signatures (AAEQ). In the generalized core/helper model defined by [90], the core can be any secure element, such as a SIM card, an NFC-based smart card, a "software-based" Trusted Execution Environment (TEE) like TrustZone or SGX, or even a TPM.

5.3 Background and Preliminaries

This section presents the considered primitives and terminologies used in describing our scheme. Note that we denote sequences and vectors in **bold**. Furthermore, by $[n]$ we denote the set of integers $\{1, \ldots, n\}$ and by $(a_i)_{i\in[L]}$, we denote the tuple $(a_1, \ldots, a_L)$.

5.3.1 Bilinear Groups and Pairings

Let $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_T$ be three finite cyclic groups with prime order p. We work with bilinear groups $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$, where e defines the mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, which is bilinear, i.e., $e(g_1^x, g_2^y) = e(g_1, g_2)^{xy}$, non-degenerate, i.e., for all generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, $e(g_1, g_2)$ generates $\mathbb{G}_T$, and efficient, i.e., there exists an efficient algorithm $\mathcal{G}(1^\lambda)$ that outputs the bilinear group and an efficient algorithm to compute $e(a, b)$ for any $a \in \mathbb{G}_1, b \in \mathbb{G}_2$. In our scheme, bilinear groups are used to support the prominent BBS+ signature scheme described in Section 5.3.4, which we utilize to express verifiable credentials as issued by some trusted issuer. Furthermore, note that pairings are often classified into one of three types (see [31] for details). However, for the purposes of this paper, it suffices to say that we consider the third type (Type-III), which allows for efficient operations in $\mathbb{G}_1$ and is used in the considered BBS+ signature scheme, which

has been proven secure under the JOC version (supporting Type-III pairings) of the q-Strong Diffie-Hellman (qSDH) assumption [24]. Finally, we use $1_{\mathbb{G}}$ to denote the identity element in the group $\mathbb{G}$.

5.3.2 Pedersen Commitment

The Pedersen commitment scheme [149] is an unconditionally hiding and computationally binding commitment scheme based on the discrete logarithm problem. It consists of three algorithms $\mathsf{Ped} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{VerCommit})$ that work as follows and satisfy the notions of *correctness*, *binding*, and *hiding* as defined below.

- $\mathsf{Ped.Setup}(1^\lambda, n) \rightarrow \mathsf{ck}$: given a security parameter and a desired number of values n, take $(h_0, \ldots, h_n) \leftarrow\!\!\$\ \mathbb{G}_1^{n+1}$, and output $\mathsf{ck} \leftarrow (h_0, \ldots, h_n)$ as the commitment key.
- $\mathsf{Ped.Commit}(\mathsf{ck}, (u_1, \ldots, u_n)) \rightarrow (t, o)$: given a commitment key and a sequence of values, parse $\mathsf{ck} = (h_0, \ldots, h_n)$, take $o \leftarrow\!\!\$\ \mathbb{Z}_q$, compute $t \leftarrow h_0^o \prod_{i=1}^n h_i^{u_i}$, and output t as the commitment, and o as the opening value (blinding factor).
- $\mathsf{Ped.VerCommit}(\mathsf{ck}, t, (u_1, \ldots, u_n), o) \rightarrow b \in \{0, 1\}$: given a commitment key, a commitment, a sequence of values, and an opening, parse $\mathsf{ck} = (h_0, \ldots, h_n)$, and only accept ($b = 1$) the commitment if $t = h_0^o \prod_{i=1}^n h_i^{u_i}$.

Correctness. For all $\lambda \in \mathbb{N}$ and any vector $\boldsymbol{u}$ of n values we have:

$$\Pr\left[\begin{matrix}\mathsf{ck} \leftarrow \mathsf{Setup}(\lambda, n) \\ (t, o) \leftarrow \mathsf{Commit}(\mathsf{ck}, \boldsymbol{u})\end{matrix} : \mathsf{VerCommit}(\mathsf{ck}, t, \boldsymbol{u}, o) = 1\right] = 1$$

Binding. For every polynomial-time adversary $\mathcal{A}$ we have:

$$\Pr\left[\begin{matrix}\mathsf{ck} \leftarrow \mathsf{Setup}(\lambda, n) \\ (t, \boldsymbol{u}, o, \boldsymbol{u}', o') \leftarrow \mathcal{A}(\mathsf{ck})\end{matrix} : \begin{matrix}\mathsf{VerCommit}(\mathsf{ck}, t, \boldsymbol{u}', o') \\ \wedge \mathsf{VerCommit}(\mathsf{ck}, t, \boldsymbol{u}, o) \\ \wedge \boldsymbol{u} \neq \boldsymbol{u}'\end{matrix}\right] = \mathsf{negl}$$

Hiding. For $\mathsf{ck} \leftarrow \mathsf{Setup}(\lambda, n)$ and every $\boldsymbol{u}, \boldsymbol{u}'$, we require $\mathsf{Commit}(\mathsf{ck}, \boldsymbol{u}) \approx \mathsf{Commit}(\mathsf{ck}, \boldsymbol{u}')$.

5.3.3 Proof of Knowledge of Algebraic Statements

To prove knowledge of the secret ingredients (u, o) of a Pedersen commitment c without disclosing either value, i.e., neither the committed value u nor its opening o, we can run a zero-knowledge proof of knowledge protocol. There are essentially two common ways to design non-interactive zero-knowledge (NIZK) proofs: Sigma protocols and zkSNARK constructions [4]. The former is highly efficient for proving algebraic statements, while the latter is superior for more expressive arithmetic representations. In our scheme, we utilize both: the latter to prove arbitrary predicates expressed as arithmetic circuits on credential attributes and the former to prove that those inputs originated from a valid credential.

When referring to zero-knowledge proofs of knowledge of discrete logarithms and statements about them, we adopt the notation of [31]. For example, $PoK\{(a, b, c) : y =$

$g^a h^b \wedge \tilde{y} = \tilde{g}^a \tilde{h}^c\}$ denotes a "*zero-knowledge proof of knowledge of integers (scalars) a, b, and c such that* $y = g^a h^b$ *and* $\tilde{y} = \tilde{g}^a \tilde{h}^c$ *holds*," where $y, g, h, \tilde{y}, \tilde{g}$, and $\tilde{h}$ are elements of some groups $\mathbb{G} = \langle g \rangle = \langle h \rangle$ and $\tilde{\mathbb{G}} = \langle \tilde{g} \rangle = \langle \tilde{h} \rangle$, respectively. The convention is that the values in the parenthesis (a, b, c) represent the secret knowledge (witnesses) that is being proven by using the other values to which the verifier has access. In our construction, we consider the following, generalized Schnorr protocol [158] to create proofs of such composite statements.

5.3.3.1 Schnorr Proof of Discrete Log

Given a protocol description in the notation in Section 5.3.3, a common method of compiling the actual protocol is following the idea behind the Schnorr *proof of knowledge of a discrete logarithm* protocol [158], which is a traditional three-move zero-knowledge Sigma protocol, i.e., a commit-challenge-response protocol. The idea is relatively simple. For proving knowledge of the value a in $y = g^a$, the prover generates randomness r and sends $t \leftarrow g^r$ to the verifier. Then, the verifier generates a random challenge c and sends it to the prover. The prover now computes the challenge response $s \leftarrow r + ca$, and sends s to the verifier. The verifier is convinced that the prover knows the discrete log of y only if $g^s = ty^c$. Furthermore, to make it a NIZK, i.e., collapse the three moves into one single move, we can use the Fiat-Shamir heuristic [69] in the random oracle model by replacing the verifier's random challenge with that of a value from a hash function H (modeled as a random oracle) on the prover's first message t and the input. Thus, in one round, the prover computes the challenge $c \leftarrow \mathsf{H}(g \| y \| t)$ and response $s \leftarrow r - cm$, and then sends $\pi = (c, s)$ to the verifier, who computes $t' \leftarrow g^s y^c$ and $c' \leftarrow \mathsf{H}(g \| y \| t')$, and accepts the proof if the challenges match: $c = c'$.

Note that we can generalize the Schnorr method to prove knowledge of the solutions (discrete logarithms) to several terms, each containing several exponents, where, for each term, $y = g^a h^a$, the prover transmits one group element and one response value for each exponent. The general idea for proving the AND (i.e., conjunction) of multiple statements is to execute them in parallel and use the same challenge. It gets more complicated for OR proofs, i.e., the disjunction of statements. In our construction, we use the idea described in [35] for composing a disjunction of statements to create designated verifier proofs.

5.3.4 BBS+ Signatures

Inspired by the group signature scheme in [25], BBS+ signatures [31] are a multi-message digital signature scheme that allows for signing an ordered list of messages where the specially produced signature has a constant size, regardless of the number of messages. In the context of verifiable credentials, note that we consider "attributes" instead of "messages", i.e., a verifiable credential is an ordered set of attributes (representing different claims) with a corresponding BBS+ signature over those attributes from some trusted issuer. Given such a BBS+ signature, the credential holder can create zero-knowledge *proofs of knowledge* of the signature and the corresponding signed attributes and optionally disclose select attributes.

In total, a BBS+ signature scheme consist of three algorithms $\mathsf{BBS+} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ that work as follows.

- $\mathsf{BBS+.KeyGen}(L) \rightarrow (\mathsf{ick}, \mathsf{ipk}, \mathsf{isk})$: given a desired number of attributes L, take $(h_0, \ldots, h_L) \leftarrow\$ \mathbb{G}_1^{L+1}$, $\mathsf{isk} \leftarrow\$ \mathbb{Z}_p^*$, $\mathsf{ipk} \leftarrow g_2^{\mathsf{isk}}$, and output $\mathsf{ick} \leftarrow (h_0, \ldots, h_L)$ as the commitment key and ipk and isk as the public and secret keys, respectively.

- $\mathsf{BBS+.Sign}(\mathsf{isk}, \mathsf{ick}, (a_1, \ldots, a_L)) \rightarrow \sigma$: given a secret key, a commitment key, a sequence of attributes to sign, parse $\mathsf{ick} = (h_0, \ldots, h_L)$, choose a random $e, s \leftarrow\$ \mathbb{Z}_p$, compute $A \leftarrow (g_1 h_0^s \prod_{i=1}^{L} h_i^{a_i})^{1/(e+\mathsf{isk})}$, and output $\sigma \leftarrow (A, e, s)$ as the multi-attribute BBS+ signature.

- $\mathsf{BBS+.Verify}(\mathsf{ipk}, \mathsf{ick}, (a_1, \ldots, a_L), \sigma) \rightarrow b \in \{0, 1\}$: given a public key, a commitment key, a sequence of attributes, and a purported signature, parse $\mathsf{ick} = (h_0, \ldots, h_L)$, $\sigma = (A, e, s)$, and accept ($b = 1$) the signature only if $e(A, \mathsf{ipk} \cdot g_2^e) = e(g_1 h_0^s \prod_{i=1}^{L} h_i^{a_i}, g_2)$.

Note that in our construction, we employ an extension of the above notation, which we describe in Section 5.5.2.1 to require the assistance of the credential holder's core element when producing presentations (i.e., proofs of knowledge of a BBS+ signature).

5.3.5 (Commit-Carrying) zkSNARKs

While Sigma protocols are efficient for algebraic statements, they are significantly slower when it comes to non-algebraic ones [4], e.g., cryptographic hash functions represented as arithmetic circuits. Fortunately, constructions called zero-knowledge Succinct Non-Interactive ARguments of Knowledge (zkSNARKs) [84] present an effective alternative approach to proving statements about functions represented as Boolean or arithmetic circuits C, which, in turn, are expressed in NP-complete languages such as Rank-1-Constraint-System (R1CS) or Quadratic Arithmetic Programs (QAPs).

In a nutshell, a zkSNARK allows the credential holder to prove that they have correctly executed an arithmetic circuit C on public input x and secret input w (called the witness), as follows. After taking C as input, a one-time setup is performed to give two public keys: an evaluation key ek and a verification key vk. The evaluation key ek enables credential holders to produce a proof π attesting to the fact that x and w satisfied C. The non-interactive proof π is zero knowledge and a *proof of knowledge*. The proof reveals nothing about u, but anyone can verify its correctness using only vk.

Furthermore, note that a credential holder is expected to supply attributes from its issued credential as secret witnesses to the arithmetic circuits to prove that its attributes satisfy some arbitrarily complex predicate. However, since a holder might cheat, we require that the holder additionally proves that the secret witness used in the proof generation matches the attribute in its issued credential. While the standard zkSNARK construction has no such capability built-in (and it is *costly* to express directly in a circuit), there, fortunately, exists an alternative construction called *commit-carrying zkSNARKs* (cc-zkSNARKs) [37], where the proof additionally contains a commitment (in our case we consider the Pedersen commitment described in Section 5.3.2) to some portion u of the witness, i.e., we assume that the witness can be split into two subdomains $w = (u, \omega)$, where ω refers to the non-committed part of the witness.

In total, cc-zkSNARK schemes consist of three algorithms $\mathsf{cc}\Pi = (\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{VerProof})$ that work as follows and satisfy the notions of *zero-knowledge*, *completeness*, *succinctness*, *knowledge soundness*, and *binding* as defined below.

- $\mathsf{cc\Pi.KeyGen}(\mathsf{C}, W, 1^\lambda) \rightarrow (\mathsf{ck}, \mathsf{ek}, \mathsf{vk})$: given an arithmetic circuit C, a desired number of witnesses to commit to W, and a security parameter λ, output a common reference string that includes a commitment key ck with $W + 1$ generators, an evaluation key ek, and a verification key vk.
- $\mathsf{cc\Pi.Prove}(\mathsf{C}, \mathsf{ek}, x, w) \rightarrow (t, \pi, o)$: given an evaluation key ek for a circuit C, public input x and secret witness $w = (u, \omega)$ such that $\mathsf{C}(x, w)$ holds, output a proof π, commitment t, and opening o such that $\mathsf{Ped.VerCommit}(\mathsf{ck}, t, u, o) = 1$.
- $\mathsf{cc\Pi.VerProof}(\mathsf{C}, \mathsf{vk}, x, t, \pi) \rightarrow b \in \{0, 1\}$: given a verification key vk for a circuit C, public input x, a commitment t, either accepts ($b = 1$) or rejects ($b = 0$) the proof π.

Completeness. For any $\lambda \in \mathbb{N}$ and C where $\mathsf{C}(x, w) = 1$, it holds:

$$\Pr\left[\begin{array}{c}(\mathsf{ck}, \mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{C}, W, \lambda) \\ (t, \pi, o) \leftarrow \mathsf{Prove}(\mathsf{C}, \mathsf{ek}, x, w)\end{array} : \mathsf{VerProof}(\mathsf{C}, \mathsf{vk}, x, t, \pi)\right] = 1$$

Binding. For every polynomial-time adversary $\mathcal{A}$ the following probability is $\mathsf{negl}(\lambda)$:

$$\Pr\left[\begin{array}{c}(\mathsf{ck}, \mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{C}, W, \lambda) \\ (t, \boldsymbol{u}, o, \boldsymbol{u}', o') \leftarrow \mathcal{A}(\mathsf{C}, \mathsf{ck}, \mathsf{ek}, \mathsf{vk})\end{array} : \begin{array}{r}\mathsf{VerCommit}(\mathsf{ck}, t, \boldsymbol{u}', o') \\ \wedge \mathsf{VerCommit}(\mathsf{ck}, t, \boldsymbol{u}, o) \\ \wedge \boldsymbol{u} \neq \boldsymbol{u}'\end{array}\right]$$

Besides these notions, knowledge-soundness informally states that we can efficiently "extract" a valid witness from proofs that pass verification. Succinctness means that proofs are of size $\mathsf{poly}(\lambda) \cdot (\lambda + \log |w|)$ and can be verified in time $\mathsf{poly}(\lambda)(\lambda + |x| + \log |w|)$. Finally, zero knowledge essentially means that proofs leak nothing about the witness. See [37] for further details.

Note that we only assume commit-carrying zkSNARKs in the formalization of our scheme. Here the commitment key depends on the relation taken by KeyGen, and a commitment is freshly created by the Prove algorithm and is tied to a single proof. However, note that the cc-SNARK lifting compiler in [37] can turn any cc-SNARK into the more versatile commit-and-prove zkSNARK version where the commitment key is relation-independent and allows for finer composition of different CP-zkSNARKs. Thus, it follows that our approach can be extended to work with modular CP-zkSNARKs.

5.3.6 Designated Verifier Proofs

Let Φ be our proof statement (e.g., proof of knowledge of a BBS+ signature). Jakobsson in [104] introduced the concept of a designated verifier, which essentially means that in a proof of Φ, we ensure that the proof can convince only a particular verifier. The idea is simple, instead of directly proving Φ, we create a transcript π of the disjunctive proof statement $\Phi \vee \phi_{Bob}$, where ϕ_{Bob} is a proof of knowledge of the designated verifier's secret key, in this case, Bob's. It essentially becomes a designated verifier proof, as Bob, the designated verifier, can always use his trapdoor to simulate a transcript without satisfying Φ. However, we can convince Bob about Φ since we can only produce a correct transcript if we satisfy Φ as we do not know Bob's secret key. Furthermore, since a third party, Cindy, cannot distinguish between a transcript where Φ holds or ϕ_{Bob} holds, she reasonably rejects the proof and thus effectively stops a transfer of the conviction.

5.4 System and Threat Model

Before delving into the protocol details, we present the considered setting and assumptions concerning the protocol participants.

5.4.1 System Model

We consider a network setting with four types of entities:

1. **Holder (Helper)** is an untrusted, computationally capable device with a credential containing some attributes/claims and initially interacts with the issuer to obtain signatures over its credential. Then, on request, the helper collaborates with its core element to generate *designated verifier proofs of knowledge* of the issuer's signature over select attributes while optionally disclosing a subset of attributes and proving arbitrary relations on the remaining undisclosed attributes.

2. **Holder (Core)** is a trusted and resource-constrained element belonging to a holder and is involved in that holder's initial credential issuance phase. Before a credential is issued to the holder's primary device (helper), the core element generates a fresh asymmetric keypair, whose public key is used in the issuer's signature over the credential to require the assistance of the core element in each presentation of the credential.

3. **Designated Verifier** is a resource-constrained and potentially dishonest entity with a certified keypair who wishes to check whether a holder's credential satisfies some predicate.

4. **Issuer** is a trusted entity with a certified keypair and is responsible for securely issuing credentials to holders, which includes: (i) verifying the correctness of the attributes claimed by a holder and (ii) guaranteeing that the involvement of the trusted core element of the specific holder is needed in the generation of credential presentations.

5.4.2 Threat Model

We assume that holders and designated verifiers are mutually distrusting. The holder assumes that the designated verifier might later attempt to profit from its credential presentations by leaking them to third parties to sell whatever information can be inferred from the underlying proof statement or disclosed attributes as being valid. Conversely, the designated verifier assumes that the holder will attempt to cheat in the proof generation by posing an invalid credential as valid or claiming that its credential satisfies the verifier's predicate when it does not.

5.4.3 Trust Model

As in [104], we assume that Cindy, a third party, will not trust Bob, a designated verifier, to have produced a proof $\pi = \Phi \vee \phi_{Bob}$, where ϕ_{Bob} is a proof of knowledge of Bob's secret key and Φ is some arbitrary proof statement (i.e., policy predicate) which the holder

(Alice) wishes to prove the truth of. We also assume that Cindy is not an observer of the communication between a credential holder and the designated verifier. This assumption holds in many use cases, especially those based on Distributed Ledger Technologies, where proofs might be stored on a blockchain. We further elaborate on this decision in Section 5.4.3.1.

5.4.3.1 On the Break of the "Strong" Designated Verifier

Note that [104] also proposed the *strong designated verifier* to essentially prevent Dave, an observer of the protocol interaction between a holder (Alice) and the designated verifier (Bob), from being convinced about the statement being proven by Alice (without Bob disclosing its secret key). They argue that we can promote a protocol to become a strong designated verifier by having Alice *probabilistically* encrypt the transcript using Bob's public key since then Bob cannot convince Dave about the decrypted message (due to the probabilistic encryption) since Bob can produce indistinguishable transcripts. Later, [154] proposed a more efficient method of achieving the same strongness property without requiring the signature to be encrypted by instead requiring Bob's secret key in the verification. However, we note here that Bob might succeed in convincing Dave if we assume that Dave initially observed the specific transcript being transmitted from Alice to Bob with the help of verifiable computation (e.g., use of zkSNARKs as described in Section 5.3.5). For example, in the first case, let c be the probabilistically encrypted transcript transferred from Alice to Bob (and observed by Dave). To convince Dave that c decrypts to π using Bob's secret key x_{Bob} (without having to disclose the key), Dave can send a decryption cipher as an arithmetic circuit $\mathsf{C}_{\mathsf{Dec}}$ to Bob, which takes an encrypted message as a public input and a decryption key as the secret witness. Then Bob can generate a zkSNARK proof that $\mathsf{C}_{\mathsf{Dec}}(c, x_{Bob}) = \pi$ and send it to Dave, who gets convinced that Alice produced the specific transcript. However, this "attack" is only possible if Dave initially observed the transmission to obtain the trusted reference value c. Therefore, in our current system, we assume that any third party, such as Dave, who is interested in receiving a conviction from dishonest designated verifiers was *not* actively observing the protocol interaction between the holder and the designated verifier.

5.4.3.2 Objectives

Let Φ be a predicate defined dynamically by a certified, designated verifier. Our protocol's overarching objectives are two-fold: (i) a designed verifier, Bob, always rejects transcripts for the proof of $\Phi \vee \phi_{Bob}$ unless Φ correctly holds on credentials issued by the trusted issuer and the core element of the corresponding credential holder was involved in producing the transcript, and (ii) a proof produced to convince a designated verifier, Bob, cannot be used later to convince another verifier, Cindy. Note that while we consider the helper potentially dishonest, a well-known problem in such a setting is that a corrupted helper can always break the privacy of an anonymous credential system, e.g., by adding identifying metadata. The core cannot check such de-anonymization attacks. Nevertheless, as in [90], we do not tolerate a malicious helper producing valid credential presentations without interacting with the core.

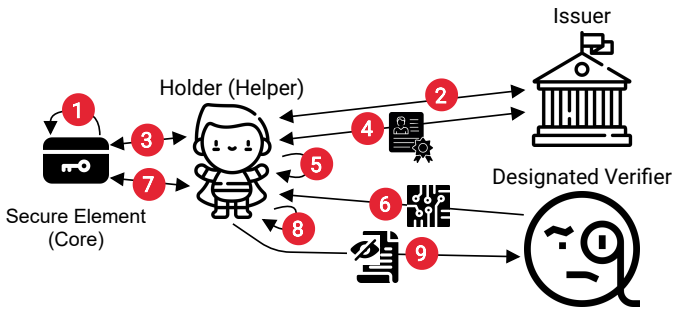


Figure 5.1: System model and conceptual work-flow.

5.5 The Protocol

5.5.1 High-Level Overview

Fig. 5.1 shows a high-level work-flow of our scheme considering the different entities described in Section 5.4.1, which follows the core/helper model of [90]. Before obtaining verifiable credentials from the issuer, the core generates a cryptographic keypair (step 1), whose secret part never leaves the shielding of the core. The holder can now request credentials from the issuer (step 2), where the core is required to supply its public key together with a *proof of knowledge* of the secret key (step 3) to ensure that the core must be involved in all presentations of the issued credentials. After the issuing (step 4), the verifiable credentials are stored on the helper (step 5). Then, to check whether a holder in the system has credentials that satisfy some arbitrarily complex predicate, designated verifiers can craft a presentation request (step 6), which includes the predicate, describes the predicate's accepted attributes, and states which attributes should be disclosed. Supposing that a holder decides to answer a presentation request, it asks for its core's contribution (step 7) before completing the proof over the predicate (step 8) and finally sending the designated-verifier proof (and any disclosed attributes) to the verifier (step 9) who either accepts or rejects the proof. Note that steps 2 to 4 must occur over authentic channels, which can be realized in multiple ways. However, in this paper, we are not interested in how the issuer verifies the holder's claims before issuing authentic credentials nor in how we can establish an authentic channel during the issuance phase, i.e., we assume ideal functionality for the channel.

5.5.2 Building Blocks

We employ the cryptographic primitives described in Section 5.3 to create our protocol, which has several resemblances to the DAA with attributes (DAA-A) protocol proposed in [31]. Specifically, authors of [31] extend the BBS+ signature scheme described in Section 5.3.4 to require the public part of a TPM-generated secret in the issuer's BBS+ signature, thus requiring the TPM's contribution whenever the platform wants to produce *proofs of knowledge* of the signature. Similarly, we consider the same extension of the BBS+ signature, which we describe in Section 5.5.2.1, to require contributions from the trusted core element of credential holders in credential presentations. However, note that

for clarity in presenting the primary objectives of this paper, we exclude the signature-based revocation and the use of pseudonyms from the protocol in [31], which we instead defer as extensions to decorate our protocol in Section 5.7.

Recall that the objective of this paper is: (i) extending the BBS+ scheme to support arbitrary predicates over attributes expressed as arithmetic circuits using cc-zkSNARKs and (ii) making credential presentations (i.e., proofs) *designated verifier*. However, note that we are working with two separate proofs. The first proof a holder must produce is a proof that its attributes satisfy the predicate using some sound cc-zkSNARK proof system, which returns, besides a proof of correctness of the predicate, a Pedersen commitment over the attributes that were passed as secret witnesses. The second proof is a *composite proof of knowledge* of a valid BBS+ signature where certain undisclosed attributes (as specified by the verifier) match the secret witnesses in the Pedersen commitment associated with the zkSNARK proof, thus proving that the attributes that satisfied the predicate originated from the issued credential. Note that both proofs must be made *designated verifier*. We describe how to make the former proof designated verifier in Section 5.5.2.2 and the latter in Section 5.5.2.3. We then put everything together in Section 5.5.4.

5.5.2.1 Split BBS+ Signatures

We denote the extended BBS+ signature scheme, wherein the issuer commits to the trusted core's public key in its BBS+ signature, as a *split* (core/helper) BBS+ scheme. The scheme consist of three algorithms $\mathsf{sBBS+} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ that work as follows. Note that the scheme's security follows [31].

- $\mathsf{sBBS+.KeyGen}(L) \rightarrow (\mathsf{ick}, \mathsf{ipk}, \mathsf{isk})$: given a desired number of attributes L, take $\mathsf{ick} \leftarrow\!\!\$\ \mathbb{G}_1^{L+2}$, $\mathsf{isk} \leftarrow\!\!\$\ \mathbb{Z}_p^*$, $\mathsf{ipk} \leftarrow g_2^{\mathsf{isk}}$, and output ick, ipk, and isk as the issuer's commitment key, public key, and secret key, respectively.

- $\mathsf{sBBS+.Sign}(\mathsf{isk}, \mathsf{ick}, \mathsf{cpk}, (a_1, \ldots, a_L)) \rightarrow \sigma$: given a secret key, commitment key, a core's public key, and a sequence of attributes to sign, choose a random $e, s \leftarrow\!\!\$\ \mathbb{Z}_p$, compute $A \leftarrow (g_1 \mathsf{ick}_0^s \mathsf{cpk} \prod_{i=1}^{L} \mathsf{ick}_{i+1}^{a_i})^{1/(e+\mathsf{isk})}$, and output $\sigma \leftarrow (A, e, s)$ as the multi-attribute BBS+ signature.

- $\mathsf{sBBS+.Verify}(\mathsf{ipk}, \mathsf{ick}, \mathsf{cpk}, (a_1, \ldots, a_L), \sigma) \rightarrow b \in \{0, 1\}$: in a public key, a commitment key, a core's public key, a sequence of attributes, and a purported signature, parse $\sigma = (A, e, s)$, and accept ($b = 1$) the signature only if $A \neq 1_{\mathbb{G}_1}$ and $e(A, \mathsf{ipk} \cdot g_2^e) = e(g_1 \mathsf{ick}_0^s \mathsf{cpk} \prod_{i=1}^{L} \mathsf{ick}_{i+1}^{a_i}, g_2)$.

Before issuing verifiable BBS+ signatures to credential holders for credentials with a certain number of attributes L, an issuer must first create its keypair: $(\mathsf{ick}, \mathsf{ipk}, \mathsf{isk}) \leftarrow \mathsf{sBBS+.KeyGen}(L)$ and register its public key and commitment key (i.e., generators in $\mathbb{G}_1$) at some trusted certificate authority as described in [31].

Assumptions. For brevity, we consider an already registered issuer. Furthermore, we assume that protocol participants have been equipped with the system parameters consisting of a security parameter λ, a bilinear group $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p with generators $g_1, \mathsf{ick}_0, \ldots, \mathsf{ick}_{L+1}$ of $\mathbb{G}_1$ and g_2 of $\mathbb{G}_2$ and a bilinear map e, generated via $\mathcal{G}(1^\lambda)$.

We further assume a random oracle $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$, which is used for the Fiat-Shamir heuristic [69] to make non-interactive zero-knowledge proofs in the random oracle model.

PoK of BBS+ Signature. Let $D \subset \{1,\ldots,L\}$ denote the selection of attribute indices that a holder wants to disclose as part of its proof of knowledge and $U = \{1,\ldots,L\} \setminus D$ denote the set of undisclosed attributes. To prove knowledge of a BBS+ signature while selectively disclosing attributes a_i with $i \in D$, the holder first computes $b \leftarrow g_1\mathbf{ick}_0^s\mathsf{cpk}\prod_{i=1}^{L}\mathbf{ick}_{i+1}^{a_i}$ and then proceeds as follows. Randomize the credential by taking $r_1 \leftarrow\!\$\ \mathbb{Z}_p^*$, set $A' \leftarrow A^{r_1}$, and set $r_3 \leftarrow 1/r_1$. Set $\overline{A} \leftarrow A'^{-e}\cdot b^{r_1}(= A'^{\mathsf{isk}})$. Choose $r_2 \leftarrow\!\$\ \mathbb{Z}_p$, set $d \leftarrow b^{r_1}\cdot\mathbf{ick}_0^{-r_2}$, and set $s' \leftarrow s - r_2\cdot r_3$. The holder now proves knowledge of a BBS+ signature following (5.1).

$$\begin{aligned}&\pi \in PoK\{(\mathsf{csk},\{a_i\}_{i\in U},e,r_2,r_3,s') :\\ &\overline{A}/d = A'^{-e}\mathbf{ick}_0^{r_2} \wedge g_1\prod_{i\in D}\mathbf{ick}_{i+1}^{a_i} = d^{r_3}\mathbf{ick}_0^{-s'}\mathbf{ick}_1^{-\mathsf{csk}}\prod_{i\in U}\mathbf{ick}_{i+1}^{-a_i}\}\end{aligned} \tag{5.1}$$

The resulting proof is $(A',\overline{A},d,\pi)$. To verify a proof, the verifier checks $A' \neq 1_{\mathbb{G}_1}$, $e(A',\mathsf{ipk}) = e(\overline{A},g_2)$, and verifies the proof π.

5.5.2.2 *Designated Verifier Circuits*

Recall that a trapdoor allows us to simulate a valid proof without knowing the satisfying witness, and the simulated proof is indistinguishable from a "real" proof. While typically, we would not want a prover to have access to a trapdoor, our scheme depends on trapdoors to make proofs *designated verifier*. Without a trapdoor, a proof that a predicate (i.e., circuit) was satisfied can be very revealing to third parties for which the proof was not originally intended. This is particularly true for circuits representing more complex functions, such as membership checks, knowledge of preimages (secrets), and range checks.

While it might be ridiculous to include a trapdoor inside simple circuits where anyone knows a satisfying witness, e.g., a circuit that only includes a range check, more complex circuits must include a trapdoor since satisfying witnesses are not publicly known. We continue by first defining designated verifier circuits in Definition 5.5.1, and then proceed by giving three examples of trapdoor functions with Bob as the contextual designated verifier. (We compare the performance of each trapdoor in our evaluation in Section 5.6.)

Definition 5.5.1 (Designated Verifier Circuit). Let C be a circuit for the function f on public input x and secret witness w, i.e., $\mathsf{C}(x,w)$ holds only if $f(x,w)$ holds. We define a *designated verifier circuit*, DVC, as a circuit that accepts $x = (x_1,x_2)$ and $w = (u,\omega)$, and includes a second function h, which we call the designated trapdoor, such that $\mathsf{DVC}(x, w)$ holds either if $f(x_1,u)$ holds *or* $h(x_2,\omega)$ holds.

Example 5.5.2 (Trapdoor #1: PoK of RSA secret key). Bob has a certified RSA keypair with some public modulus n. We can define our designated trapdoor h in our circuit $\mathsf{DVC}($ $(x_1,x_2),(u,\omega))$ as: $h(n,p,q) : p\times q = n$, where p and q are passed as secret witnesses to the circuit in ω and n is passed in x_2. Thus, anyone besides Bob who sees a proof with n as a public input will reject it since Bob might have cheated. Conversely, anyone except Bob can only create a valid proof over the circuit with n in x_2 if they know a satisfying witness to $f(x_1,u)$ since they do not know Bob's secret key.

Example 5.5.3 (Trapdoor #2: PoK of EC secret key). Bob has a certified EC keypair with public key $y = g^x$. We can define our designated trapdoor h in our circuit $\mathsf{DVC}((x_1, x_2), (u, \omega))$ as: $h(y, x) : y = g^x$, where x is passed as a secret witness to the circuit in ω and y is passed in x_2. The conviction follows that of example 5.5.2.

Example 5.5.4 (Trapdoor #3: PoK of preimage). Bob has a secret value x with a certified image $y = \mathsf{H}(x \,\|\, r)$, where H is a sound hashing function, and r is some secret blinding factor initially supplied by Bob. We can define our designated trapdoor h in our circuit $\mathsf{DVC}((x_1, x_2), (u, \omega))$ as: $h(y, x, r) : y = \mathsf{H}(x \,\|\, r)$, where x and r are passed as secret witnesses to the circuit in ω and y is passed in x_2. The conviction follows that of example 5.5.2.

In practice, we need assurance that a predicate includes a trapdoor to which the designated verifier has access. For example, consider a circuit C with a supposed trapdoor from Example 5.5.3 and includes ek and vk as its evaluation and verification keys, respectively. We can determine the existence of this trapdoor with (5.2).

$$\begin{aligned} (\mathsf{pk}', \mathsf{sk}') &\leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda) \\ (t', \pi', o') &\leftarrow \mathsf{cc\Pi.Prove}(\mathsf{C}, \mathsf{ek}, (x, \mathsf{pk}'), (\{0,1\}^W, \mathsf{sk}')) \\ &\qquad \mathsf{cc\Pi.VerProof}(\mathsf{C}, \mathsf{vk}, (x, \mathsf{pk}'), t', \pi') \stackrel{?}{=} 1 \end{aligned} \tag{5.2}$$

If (5.2) holds, the circuit is designated verifier according to Definition 5.5.1. However, note that such checks require an extra execution of the proof generation for a circuit, which can be expensive for large circuits. Thus, in practice, we might outsource such trapdoor verification either to a certification authority or distributed worker farms [182] that return proof about the presence of the trapdoor. For brevity, in the remaining paper, we denote by DVC a circuit that has been verified to be a designated verifier circuit.

5.5.2.3 Designated Verifier Sigma Protocols

Let us assume that we have a credential comprising L attributes: $(a_1, \ldots, a_L)$. Let D denote the attribute indices we are supposed to disclose. The remaining undisclosed attribute indices are $U = \{1, \ldots, L\} \setminus D$. Furthermore, let DVC be a circuit for which we want to prove to the designated verifier that we have satisfying witnesses. Finally, let $\boldsymbol{u} = (u_1, \ldots, u_W)$ specify the sequence of attribute indices that we must supply as witnesses to the circuit, where $\boldsymbol{u}_i \in U$ and W denotes the slice of committed witnesses as determined during the circuit's key generation, i.e., the circuit's commitment key ck contains $W + 1$ generators.

To produce a designated-verifier proof over the circuit using the specified attributes from our credential, where dvpk is the public key of the designated verifier, and x is some public input, we run:

$$(t_u, \pi, o) \leftarrow \mathsf{cc\Pi.Prove}(\mathsf{DVC}, \mathsf{ek}, (x, \mathsf{dvpk}), ((a_{\boldsymbol{u}_i})_{i \in [W]}, \omega))$$

To prove knowledge of the secret ingredients of the Pedersen commitment while also allowing the designated verifier to produce valid proofs, we form the following disjunc-

tive proof statement:

$$\pi \in PoK\{(\{a_i\}_{i\in \boldsymbol{u}}, o) \vee \mathsf{dvsk} : t_u = \mathsf{ck}_0^o \prod_{i=1}^{W} \mathsf{ck}_i^{a_{\boldsymbol{u}_i}} \vee \mathsf{dvpk} = g^{\mathsf{dvsk}}\} \tag{5.3}$$

Then, to produce non-interactive proofs that we know of a solution to one of the problems without anyone learning which solution we know, we use the idea described in [35], which is based on the initial result proposed by Cramer et al. [51]. The idea is based on the generalization of the Schnorr protocol [158] (described in Section 5.3.3.1), where we execute the two proof systems in parallel, but we additionally allow the prover to "cheat" in one of them in an indistinguishable manner, as follows (computations are done modulo the curve order p). Pick $r_{a_i} \leftarrow\!\!\$\ \mathbb{Z}_p$ for $i \in \boldsymbol{u}$, $r_o, r_{\mathsf{dvsk}} \leftarrow\!\!\$\ \mathbb{Z}_p$, and a random (cheating) challenge $c_2 \leftarrow\!\!\$\ \mathbb{Z}_p$ for the second term of (5.3) since we are not the designated verifier. We then compute the two commitments: $t_1 \leftarrow \mathsf{ck}_0^{r_o} \prod_{i=1}^{W} \mathsf{ck}_i^{r_{a_{\boldsymbol{u}_i}}}$ and $t_2 \leftarrow g^{r_{\mathbf{dvsk}}} \cdot \mathsf{dvpk}^{c_2}$. To get the challenge for the first term, c_1, we first compute $c \leftarrow \mathsf{H}(t_1 \,\|\, t_2 \,\|\, \mathsf{ck} \,\|\, g \,\|\, t_u \,\|\, \mathsf{dvpk})$ and then $c_1 \leftarrow c - c_2$. We can then compute the challenge responses as: $s_{a_i} \leftarrow r_{a_i} + c_1 a_i$ for $i \in \boldsymbol{u}$, $s_o \leftarrow r_o + c_1 o$, and $s_{r_{\mathbf{dvsk}}} \leftarrow r_{\mathsf{dvsk}}$. Our final proof transcript for (5.3) is $\pi' \leftarrow (c_1, c_2, t_u, \{s_{a_i}\}_{i\in \boldsymbol{u}}, s_o, s_{r_{\mathbf{dvsk}}})$, which we give to the designated verifier together with the zkSNARK proof π.

To verify that π' is a valid transcript for (5.3), where $t_u \in \pi'$ is a satisfying commitment to π, i.e., $\mathsf{cc}\Pi.\mathsf{VerProof}(\mathsf{DVC}, \mathsf{vk}, (x, \mathsf{dvpk}), t_u, \pi) = 1$ holds, the verifier first reconstructs the commitments: $t_1' \leftarrow \mathsf{ck}_0^{s_o} \cdot (\prod_{i=1}^{W} \mathsf{ck}_i^{s_{a_{\boldsymbol{u}_i}}}) \cdot t_u^{-c_1}$ and $t_2' \leftarrow g^{s_{r_{\mathbf{dvsk}}}} \cdot \mathsf{dvpk}^{c_2}$, and then checks that $c_1 + c_2 = \mathsf{H}(t_1' \,\|\, t_2' \,\|\, \mathsf{ck} \,\|\, g \,\|\, t_u \,\|\, \mathsf{dvpk})$ holds. Note that the only missing ingredient of (5.3) is proving that the attributes originated from a valid BBS+ signature, i.e., that the discrete logarithms of the Pedersen commitment t_u match the attribute indices specified by $\boldsymbol{u}$ in a valid credential, which we explain in Section 5.5.4.

5.5.3 Core/Helper Credential Issuance

As described in Section 5.5.2.1, we assume that protocol participants have been equipped with the system parameters and the issuer's trusted cryptographic materials. Fig. 5.2 shows the issuance protocol initiated by a holder that wishes to get the issuer's BBS+ signature over its claims. After the issuer has performed all the necessary steps to verify the authenticity of the holder's claims, it opens a communication channel with the holder's trusted core element. Here the issuer ensures the core's active participation whenever the helper wishes to produce *proofs of knowledge* of the issuer's BBS+ signature over the holder's claims. To do so, the issuer sends a fresh challenge to the core, requesting the core to generate a fresh key pair whose public key should be included in the BBS+ signature. Note that by including the core's public key in the signature, we prevent the helper from independently producing *proofs of knowledge* of the signature since only the core can produce the necessary contributions in the Schnorr protocol to prove knowledge of the public key's discrete logarithm, i.e., the core secret key csk in (5.1). Like [31], we assume that this communication between the issuer and the core occurs over an authenticated channel and, similarly, that there is a secure channel between the holder and its core.

When the core has created its key pair, it produces a *signed proof of knowledge* that it knows the secret key behind its public key: $\pi \leftarrow\!\!\$\ SPoK\{(\mathsf{csk}) : \mathsf{cpk} = \mathsf{ick}_1^{\mathsf{csk}}\}(n)$, where

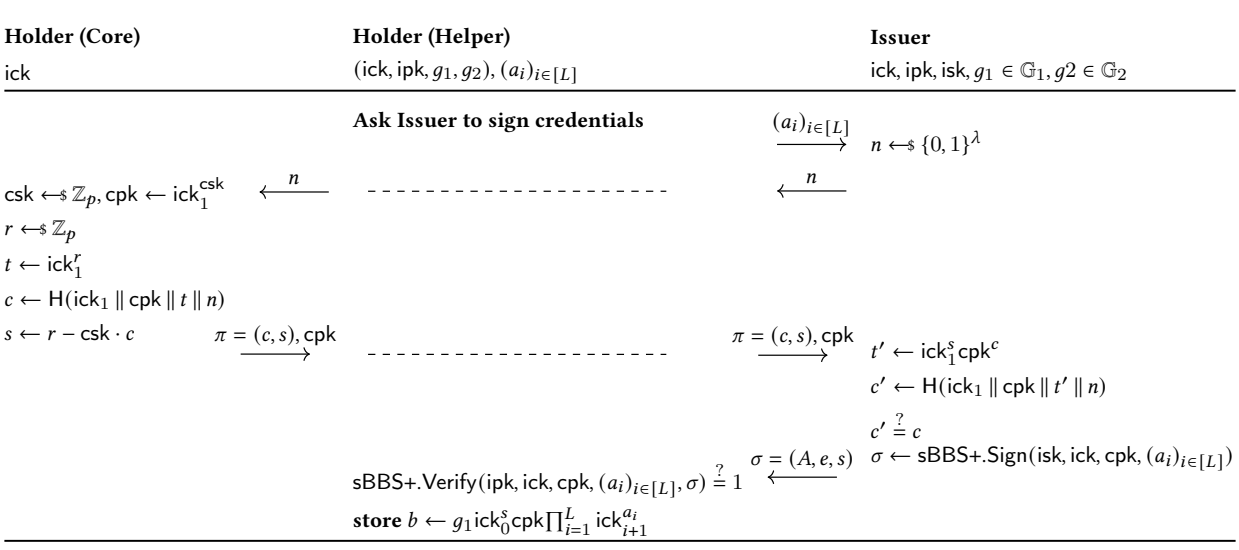


Figure 5.2: Credential issuance using the split BBS+ signature scheme.

n is the verifier's fresh challenge. The proof and public key are then sent to the issuer, who, after verifying the proof, proceeds to use its secret key to generate a BBS+ signature over the holder's claims that also incorporates the core's public key. Finally, given the issuer's signature, the holder checks the signature's validity before storing it in persistent storage. The holder also stores the product the issuer signed to ease the computational effort needed to produce credential presentations.

5.5.4 Designated Verifier Credential Presentations

To demonstrate the protocol, let us consider a simple setting where a credential holder knows of a designated verifier, and the designated verifier wants to determine whether the holder has valid credentials whose attributes satisfy some designated verifier circuit DVC. Like Section 5.5.2.3, we consider credentials with L attributes, where the order and meaning of the attributes are known[1]. Let D denote the attribute indices the designated verifier wants to be disclosed such that $U = \{1, \ldots, L\} \setminus D$ contains the remaining undisclosed attribute indices. Finally, let $\boldsymbol{u} = (u_1, \ldots, u_W)$ specify the sequence of attribute indices that should be passed as circuit witnesses, with $\boldsymbol{u}_i \in U$. Fig. 5.3 shows the participants engage in the protocol.

Like in Section 5.5.2.3, to prove that the specified attributes satisfy the predicate, the holder generates a commit-carrying zkSNARK proof over the circuit with the specified selection of attributes as the *committed* part of the witness:

$$(t_u, \pi', o) \leftarrow \mathsf{cc}\Pi.\mathsf{Prove}(\mathsf{DVC}, \mathsf{ek}, (x, \mathsf{dvpk}), ((a_{\boldsymbol{u}_i})_{i\in[W]}, \omega))$$

Then, to generate a designated-verifier zero-knowledge proof of a valid BBS+ signature while proving that the committed witnesses t_u from the zkSNARK proof equal the

[1] In practice, the credentials follow richer and standardized formats (e.g., the schema defined by W3C [179]), where we encounter additional challenges, such as secure serialization.

Holder (Core)		**Holder (Helper)**		**Designated Verifier**
$\mathsf{ick}, (\mathsf{csk}, \mathsf{cpk})$		$(\mathsf{ick}, \mathsf{ipk}, g_1, g_2), (a_i)_{i\in[L]}, \sigma = (A, e, s), b, \mathsf{dvpk}$		$(\mathsf{dvsk}, \mathsf{dvpk} = g^{\mathsf{dvsk}}), (\mathsf{ick}, \mathsf{ipk}, g_1, g_2), (\mathsf{DVC}, \mathsf{ck}, \mathsf{ek}, \mathsf{vk}), x$
				$D \subset \{1, \ldots, L\}$ **attribute indices to disclose**
		Generate cc-zkSNARK proof		$\boldsymbol{u} = (u_1, \ldots, u_W)$ **indices to pass as witness to circuit**
		$w \leftarrow ((a_{\boldsymbol{u}_i})_{i\in[W]}, \omega)$	$(\mathsf{DVC}, \mathsf{ck}, \mathsf{ek}), D, \boldsymbol{u}, x, n \longleftarrow$	$n \leftarrow\!\$\ \{0,1\}^\lambda$
		$(t_u, \pi', o) \leftarrow \mathsf{cc\Pi.Prove}(\mathsf{DVC}, \mathsf{ek}, (x, \mathsf{dvpk}), w)$		
		Randomize BBS+ credential		
		$r_1 \leftarrow\!\$\ \mathbb{Z}_p^*, r_2 \leftarrow\!\$\ \mathbb{Z}_p, r_3 \leftarrow \frac{1}{r_1}$		
		$A' \leftarrow A^{r_1}$		
		$\overline{A} \leftarrow A'^{-e} \cdot b^{r_1}$		
		$d \leftarrow b^{r_1} \cdot \mathsf{ick}_0^{-r_2}$		
		$s' \leftarrow s - r_2 r_3$		
		Schnorr PoK of discrete logarithms		
		$r_{a_i} \leftarrow\!\$\ \mathbb{Z}_p$ for $i \in U = \{1, \ldots, L\} \setminus D$		
		$r_e, r_{r_2}, r_{r_3}, r_{s'}, r_o, r_{\mathsf{dvsk}}, c_2 \leftarrow\!\$\ \mathbb{Z}_p$		
$r_{\mathsf{csk}} \leftarrow\!\$\ \mathbb{Z}_p$	$\longleftarrow$	**get Schnorr commitment for secret key**		
$t_{\mathsf{csk}} \leftarrow \mathsf{ick}_1^{r_{\mathsf{csk}}}$	$\overset{t_{\mathsf{csk}}}{\longrightarrow}$			
		$t_1 \leftarrow A'^{r_e} \cdot \mathsf{ick}_0^{r_{r_2}}$		
		$t_2 \leftarrow d^{r_{r_3}} \cdot \mathsf{ick}_0^{r_{s'}} \cdot t_{\mathsf{csk}} \prod_{i\in U} \mathsf{ick}_{i+1}^{r_{a_i}}$		
		$t_3 \leftarrow \mathsf{ck}_0^{r_o} \cdot \prod_{i=1}^{W} \mathsf{ck}_i^{r_{a_{\boldsymbol{u}_i}}}$		
		$t_4 \leftarrow g^{r_{\mathsf{dvsk}}} \cdot \mathsf{dvpk}^{c_2}$		
		$c \leftarrow \mathsf{H}(n \,\|\, A' \,\|\, \overline{A} \,\|\, d \,\|\, t_1 \,\|\, t_2 \,\|\, t_3 \,\|\, t_4 \,\|\, t_u \,\|$ $g \,\|\, g_1 \,\|\, D \,\|\, \boldsymbol{u} \,\|\, \mathsf{ck} \,\|\, \mathsf{ick} \,\|\, \mathsf{ipk} \,\|\, \mathsf{dvpk})$		
		$c_1 \leftarrow c - c_2$		
$s_{\mathsf{csk}} \leftarrow r_{\mathsf{csk}} + c_1 \cdot \mathsf{csk}$	$\overset{c_1}{\longleftarrow}$	**get Schnorr response for secret key**		
	$\overset{s_{\mathsf{csk}}}{\longrightarrow}$			
		$s_{a_i} \leftarrow r_{a_i} + c_1 a_i$ for $i \in U$		
		$s_e \leftarrow r_e + c_1 e$		
		$s_{r_2} \leftarrow r_{r_2} - c_1 r_2$		
		$s_{r_3} \leftarrow r_{r_3} - c_1 r_3$		
		$s_{s'} \leftarrow r_{s'} + c_1 s'$		
		$s_o \leftarrow r_o + c_1 o$		
		$s_{r_{\mathsf{dvsk}}} \leftarrow r_{\mathsf{dvsk}}$		
		$\pi \leftarrow (c_1, c_2, \pi', t_u, s_{\mathsf{csk}}, \{s_{a_i}\}_{i\in U}, s_e, s_{r_2}, s_{r_3},$ $s_{s'}, s_o, s_{r_{\mathsf{dvsk}}})$	$\{A', \overline{A}, d, \pi, \{a_i\}_{i\in D}\}_{\mathsf{dvpk}} \longrightarrow$	**Verify that** $\mathsf{cc\Pi.VerProof}(\mathsf{DVC}, \mathsf{vk}, (x, \mathsf{dvpk}), t_u, \pi') \overset{?}{=} 1$
				$t_1' \leftarrow A'^{s_e} \cdot \mathsf{ick}_0^{s_{r_2}} \cdot (\overline{A}/d)^{c_1}$
				$t_2' \leftarrow d^{s_{r_3}} \cdot \mathsf{ick}_0^{s_{s'}} \cdot \mathsf{ick}_1^{s_{\mathsf{csk}}} \cdot (\prod_{i\in U} \mathsf{ick}_{i+1}^{s_{a_i}}) \cdot (g_1 \prod_{i\in D} \mathsf{ick}_{i+1}^{a_i})^{c_1}$
				$t_3' \leftarrow \mathsf{ck}_0^{s_o} \cdot (\prod_{i=1}^{W} \mathsf{ck}_i^{s_{a_{\boldsymbol{u}_i}}}) \cdot t_u^{-c_1}$
				$t_4' \leftarrow g^{s_{r_{\mathsf{dvsk}}}} \cdot \mathsf{dvpk}^{c_2}$
				$c' \leftarrow \mathsf{H}(n \,\|\, A' \,\|\, \overline{A} \,\|\, d \,\|\, t_1' \,\|\, t_2' \,\|\, t_3' \,\|\, t_4' \,\|\, t_u \,\|$ $g \,\|\, g_1 \,\|\, D \,\|\, \boldsymbol{u} \,\|\, \mathsf{ck} \,\|\, \mathsf{ick} \,\|\, \mathsf{ipk} \,\|\, \mathsf{dvpk})$
				Verify that $c' \overset{?}{=} c_1 + c_2$, $A' \overset{?}{\neq} 1_{\mathbb{G}_1}$ **and** $e(A', \mathsf{ipk}) \overset{?}{=} e(\overline{A}, g_2)$

Figure 5.3: Holder proves knowledge of a valid BBS+ signature whose undisclosed attributes satisfy some arbitrary predicate only to a designated verifier. Since the designated verifier can also create the proof using the trapdoors, it is worthless to anyone else.

attributes specified by $\boldsymbol{u}$ in the signed credential, we merge the proof statements of (5.1) and (5.3), resulting in (5.4). The holder first randomizes its BBS+ signature as described

in Section 5.5.2.1 and then produces a proof of (5.4):

$$\begin{aligned}\pi \in SPoK\Big\{(\mathsf{csk}, \{a_i\}_{i\in U}, e, r_2, r_3, s', o) \vee \mathsf{dvsk} : \Big(\\ \overline{A}/d = A'^{-e}\mathsf{ick}_0^{r_2} \wedge g_1 \prod_{i\in D} \mathsf{ick}_{i+1}^{a_i} = d^{r_3}\mathsf{ick}_0^{-s'}\mathsf{ick}_1^{-\mathsf{csk}} \prod_{i\in U} \mathsf{ick}_{i+1}^{-a_i} \\ \wedge\, t_u = \mathsf{ck}_0^{o} \prod_{i=1}^{W} \mathsf{ck}_i^{a_{\boldsymbol{u}_i}}\Big) \vee \Big(\mathsf{dvpk} = g^{\mathsf{dvsk}}\Big)\Big\}(n)\end{aligned} \tag{5.4}$$

This statement is a disjunction of two outer relations. The first outer relation is a conjunction of three inner relations, which the holder attempts to prove together with its core, and the second is for the designated verifier. Note that we create the outer disjunction following the idea in Section 5.5.2.3. Specifically, we execute the Schnorr proof of knowledge protocol for each of the four relations to prove knowledge of the different discrete logarithms. However, we use a different half of the challenge in each of the outer relations.

Finally, to verify that π is a valid transcript for (5.4), where t_u is a satisfying commitment to π', i.e., $\mathsf{cc}\Pi.\mathsf{VerProof}(\mathsf{DVC}, \mathsf{vk}, (x, \mathsf{dvpk}), t_u, \pi') = 1$ holds, the verifier reconstructs and verifies the four commitments as shown in Fig. 5.3. Then, to verify the accompanying, randomized BBS+ signature $(A', \overline{A}, d)$ against the issuer's public key, the verifier checks that $A' \neq 1_{\mathbb{G}_1}$ and $e(A', \mathsf{ipk}) = e(\overline{A}, g_2)$.

Note that if the holder attempts to use an invalid sequence of attributes to satisfy the predicate (i.e., one not specified by $\boldsymbol{u}$), then the proof will be rejected. Specifically, since the verifier's reconstruction of the t_3' commitment considers the s-values corresponding to the correct sequence of attributes as specified by $\boldsymbol{u}$, the reconstructed commitment would inevitably differ from t_u, producing a different challenge c' and thus causing the proof to be rejected.

5.5.4.1 *Simulating Transcripts*

Demonstrating how the protocol is *designated verifier*, Fig. 5.4 shows the designated verifier producing valid transcripts for arbitrary attributes using the demonstrative trapdoor in the circuit and satisfying the latter outer relation in (5.4).

5.6 Performance Evaluation

Our evaluation aims to answer the questions of (i) how efficient our protocol is for creating credential presentations and (ii) how costly the considered method is for making it *designated verifier*.

5.6.1 Implementation and Experimental Setup

To program demonstrative circuits, we used xJsnark [114], a high-level code-to-circuit compilation framework that employs a mix of optimizations to minimize circuit complexity. With xJsnark, our high-level code is compiled into low-level circuits, which, using the jsnark interface [113], are translated into the R1CS constraint system and fed

Simulate Transcript$((A', \overline{A}, d), (\mathsf{ick}, \mathsf{ipk}, g_1), (\mathsf{DVC}, \mathsf{ck}), D, \boldsymbol{u}, x, n)$

$(a_1, \ldots, a_L) \leftarrow\!\$\ \mathbb{Z}_p^L$

$w \leftarrow ((a_{\boldsymbol{u}_i})_{i \in [W]}, \mathsf{dvsk})$

$(t_u, \pi', o) \leftarrow \mathsf{cc\Pi.Prove}(\mathsf{DVC}, \mathsf{ek}, (x, \mathsf{dvpk}), w)$

$r_{a_i} \leftarrow\!\$\ \mathbb{Z}_p$ for $i \in U = \{1, \ldots, L\} \setminus D$

$r_e, r_{r_2}, r_{r_3}, r_{s'}, r_{\mathsf{csk}}, r_o, r_{\mathsf{dvsk}}, c_1 \leftarrow\!\$\ \mathbb{Z}_p$

$t_1 \leftarrow A'^{r_e} \cdot \mathsf{ick}_0^{r_{r_2}} \cdot (\overline{A}/d)^{c_1}$

$t_2 \leftarrow d^{r_{r_3}} \cdot \mathsf{ick}_0^{r_{s'}} \cdot \mathsf{ick}_1^{r_{\mathsf{csk}}} \cdot (\prod_{i \in U} \mathsf{ick}_{i+1}^{r_{a_i}}) \cdot (g_1 \prod_{i \in D} \mathsf{ick}_{i+1}^{a_i})^{c_1}$

$t_3 \leftarrow \mathsf{ck}_0^{r_o} \cdot (\prod_{i=1}^{W} \mathsf{ck}_i^{r_{a_{\boldsymbol{u}_i}}}) \cdot t_u^{-c_1}$

$t_4 \leftarrow g^{r_{\mathsf{dvsk}}}$

$c \leftarrow \mathsf{H}(n \,\|\, A' \,\|\, \overline{A} \,\|\, d \,\|\, t_1 \,\|\, t_2 \,\|\, t_3 \,\|\, t_4 \,\|\, t_u \,\|$
$g \,\|\, g_1 \,\|\, D \,\|\, \boldsymbol{u} \,\|\, \mathsf{ck} \,\|\, \mathsf{ick} \,\|\, \mathsf{ipk} \,\|\, \mathsf{dvpk})$

$c_2 \leftarrow c - c_1$

$s_{a_i} \leftarrow r_{a_i}$ for $i \in U$

$(s_e, s_{r_2}, s_{r_3}, s_{s'}, s_{\mathsf{csk}}, s_o) \leftarrow (r_e, r_{r_2}, r_{r_3}, r_{s'}, r_{\mathsf{csk}}, r_o)$

$s_{\mathsf{dvsk}} \leftarrow r_{\mathsf{dvsk}} - c_2 \cdot \mathsf{dvsk}$

$\pi \leftarrow (c_1, c_2, \pi', t_u, s_{\mathsf{csk}}, \{s_{a_i}\}_{i \in U}, s_e, s_{r_2}, s_{r_3}, s_{s'}, s_o, s_{r_{\mathsf{dvsk}}})$

return $(A', \overline{A}, d, \pi, \{a_i\}_{i \in D})$

Figure 5.4: Designated verifier simulating correct transcripts.

into the libsnark [119] backend for instantiating a particular zkSNARK proof system over the circuit. In our case, we considered an implementation [130] of the LegoGro16 [37] proof system (over the BN254 curve) for producing commit-carrying zkSNARK proofs. The specific LegoGro16 implementation is essentially a commit-and-prove variant of libsnark's implementation of Groth16 [84]. However, since we only require the zkSNARK to be commit-carrying, we assume the complexity of the commit-carrying variant here. As our testbed (holder), we considered a machine with an AMD Ryzen 7 3700X processor and 16 GB of memory (experiments were conducted in a WSL2 environment).

5.6.2 Notation

By $k\mathbb{G}_i$, we denote k exponentiations (scalar multiplications) in the group $\mathbb{G}_i$, by $k\mathbb{G}_i^j$ we denote k j-multi exponentiations, and by kP we denote k pairing operations. We let W denote the number of witnesses committed to in the considered commit-carrying zkSNARK proof system, L denote the number of attributes, and D and U denote the

Table 5.1: Comparison of our scheme's complexity to similar schemes when creating credential presentations by the holder (helper) and its secure element (core) and verification by the verifier.

Scheme	Core	Holder (helper)	Verifier	Credential size	Presentation size
DAA-A [31]	$3\mathbb{G}_1$	$O(U\mathbb{G}_1)$	$O(L\mathbb{G}_1) + 2P$	$2\mathbb{Z}_p + 1\mathbb{G}_1$	$O(U\mathbb{Z}_p) + 4\mathbb{G}_1 + 1\lambda$
DAA-A [30]	$3\mathbb{G}_1$	$O(U\mathbb{G}_1)$	$O(L\mathbb{G}_1) + 2P$	$2\mathbb{Z}_p + 1\mathbb{G}_1$	$O(U\mathbb{Z}_p) + 4\mathbb{G}_1 + 1\lambda$
CHAC [90]	$1\mathbb{G}_1$	$O(D(\mathbb{G}_1 + \mathbb{G}_2))$	$O(DP)$	$O(L(\mathbb{G}_1 + \mathbb{G}_2))$	$6\mathbb{G}_1 + 3\mathbb{G}_2$
This work (sBBS+)	$1\mathbb{G}_1$	$O(U\mathbb{G}_1)$	$O(L\mathbb{G}_1) + 2P$	$2\mathbb{Z}_p + 1\mathbb{G}_1$	$O(U\mathbb{Z}_p) + 3\mathbb{G}_1 + 2\lambda$
This work (ccGroth16)		$O(2W\mathbb{G}_1) + O((3N + M)\mathbb{G}_1 + N\mathbb{G}_2)$	$O(W\mathbb{G}_1) + 3P$		$3\mathbb{G}_1 + 1\mathbb{G}_2$

number of disclosed and undisclosed attributes, respectively[2]. Finally, to reason about the computational complexity incurred by varying arithmetic circuit sizes, we consider circuits as R1CS instances where we use M to denote the number of constraints and N to denote the number of variables in the instance.

5.6.3 Asymptotic Performance

Table 5.1 shows the computational complexity of our construction described in Section 5.5.4 and includes a comparison with other related core/helper credential protocols as described in Section 5.2. Note, however, that *none* of the other schemes consider *designated verifiers* nor zkSNARKs. Therefore, we split our protocol's effort over two rows for comparison purposes.

The difference between ours and the two DAA-A schemes is that we are not using pseudonyms and we have two challenges (for the disjunction). Like CHAC [90], our core only needs to compute a single EC scalar multiplication, regardless of the number of attributes. Furthermore, note that for the considered curve, Groth16's proof is 127 bytes and contains 3 group elements (2 $\mathbb{G}_1$ elements and 1 $\mathbb{G}_2$ element), and 3 pairings dominate verification. However, with the considered commit-carrying variant, we additionally have a Pedersen commitment to the W witnesses in $\mathbb{G}_1$.

5.6.4 Empirical Performance

To determine the cost of the different trapdoors mentioned in Section 5.5.2.2, we used xJsnark to generate the corresponding circuits. As also reported at [5], it costs only 2578 constraints to express a function for verifying knowledge of the two secret prime factors of a 2048-bit RSA key's modulus as an arithmetic circuit. However, proving knowledge of an ECDSA secret key costs almost 700K constraints if we consider the NIST P-256 curve. For the third choice of a trapdoor, we mentioned that we could express a hashing function in the circuit and then prove knowledge of the secret preimage. In our case, we considered the recent permutation function called Poseidon [82] that was made to be expressed inexpensively in an arithmetic circuit. With our implementation of Poseidon, it cost only 241 constraints when considering an arity of 2 and 262 constraints for an arity of 3, which allows for preimages fitting three field elements, i.e., a preimage bitwidth of 762 bits considering the BN254 curve. Note that from our timings of generating proofs with libsnark, it was evident that we prove the satisfaction of circuits at a rate of ≈ 77.8

[2]Following the notation used in https://github.com/scipr-lab/libsnark/tree/master/libsnark/zk_proof_systems/ppzksnark

constraints/ms on our considered setup. Finally, computing the three pairings during proof verification took approximately 2 ms using libsnark.

5.7 Security Properties and Extensions

Besides its secure implementation, the proposed RETRACT scheme's foundational security is guaranteed by the security of the underlying BBS+ signature scheme [31] and zkSNARK proof system [84, 37]. In the following, we give an intuitive description of the different security properties our scheme is designed to provide and how it achieves them. Note that we only cover essential properties here.

Property 5.7.1 (Unforgeability). As described in Section 5.5.4, the verifier will reject a proof if the holder uses attributes whose indices were not specified by $\boldsymbol{u}$. Thus, the holder cannot cheat by choosing a different sequence of attributes to satisfy the predicate. Furthermore, since a verifier only accepts a proof of (5.4) if the corresponding randomized signature is valid under the issuer's public key, only the actual holder of a credential issued by the trusted issuer can produce correct transcripts that a verifier accepts. Therefore, an entity cannot use forged or otherwise invalid credentials to convince a verifier about satisfying a predicate. Nor is it possible for an entity to present attributes that it does not possess believably.

Property 5.7.2 (Designated Verifier). Only the designated verifier can be convinced by a proof produced by a holder since the verifier can produce indistinguishable transcripts, as demonstrated in Fig. 5.4. While malicious verifiers might attempt to infringe on the privacy of credential holders by crafting predicates to lure out excessive information, we note that proofs are always limited in convincing a particular verifier. Furthermore, similar to the revocation of credentials, it is possible to castrate such misbehaving verifiers in an established credential system, or we could slightly modify the system setup to require all predicates to be certified by trusted parties.

Property 5.7.3 (Unlinkability & Selective Disclosure). Inherent to anonymous credentials, different credential presentations from the same holder should not be linkable, and it should be possible to disclose attributes in a verifiable manner selectively. Both of these properties are guaranteed by the underlying BBS+ signature scheme.

Property 5.7.4 (Dependability). Like [90], regardless of whether credentials are leaked or a helper device is completely compromised, it should only be possible to produce valid credential presentations with assistance from the trusted core associated with the holder for which the credentials were initially issued. This property is guaranteed by the split BBS+ signature scheme described in Section 5.5.2.1.

Extension 5.7.5 (Revocation). An important feature to effectively handle the dynamic nature of the set of entities of a credential system is the possibility of revoking the credentials of misbehaving parties. While not explicitly considered in our construction, there are several ways to enforce the revocation of specific credentials. For example, leveraging the expressiveness of circuits, we can include a non-membership check (e.g., using Merkle trees or RSA accumulators) and add a conjunctive clause to prevent all revoked credentials from satisfying the predicate. Another method to enforce revocation

is to require holders to produce separate non-revocation proofs when presenting their credentials, e.g., using the signature-based revocation mechanism of [31].

5.8 Conclusions

We presented RETRACT, a novel and fully expressive anonymous credential scheme allowing credential holders to prove knowledge of satisfying credentials to arbitrary predicates while ensuring that only the designated verifier believes them. Our construction demonstrated how to combine state-of-the-art commit-carrying zkSNARK constructions with the widely used BBS+ signature scheme.

Chapter 6

Conclusions

In this thesis, we have explored how an advanced attacker can remotely infect resource-constrained embedded systems, like the Tmote Sky module, and systematically extract data from memory, which, in certain circumstances, might include cryptographic keys, without assuming any knowledge about the target system's software. We also proposed three privacy-preserving protocols: ZEKRO, a remote attestation scheme, wherein a prover equipped with a Trusted Platform Module can continuously convince an untrusted verifier that its platform configuration is correct without disclosing any concrete state details; ZEKRA, a Control-Flow Attestation scheme that utilizes Verifiable Computation based on zkSNARKs to let a prover convince an untrusted verifier that it executed a specific program correctly according to its Control-Flow Graph (CFG), i.e., in the absence of runtime attacks (e.g., Return-Oriented Programming attacks); and RETRACT, an anonymous credential scheme that lets a credential holder convince a designated verifier that it holds a valid credential that satisfies arbitrary criteria by utilizing Verifiable Computation based on commit-carrying zkSNARKs for facilitating expressiveness, BBS+ signatures for creating the credentials, and trapdoors for making all proofs designated-verifier. By using Trusted Computing and Privacy-Enhancing Technologies (PETs), these protocols allow a prover to convince an untrusted verifier of the correctness or validity of a platform, program execution, or credential, without disclosing any specific details.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[3] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems. In *NDSS*, 2019.

[4] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. Non-interactive zero-knowledge proofs for composite statements. In *Annual International Cryptology Conference*, pages 643–673. Springer, 2018.

[5] akosba. akosba/xjsnark: A high-level framework for developing efficient zk-snark circuits. Retrieved December 20, 2022 from https://github.com/akosba/xjsnark.

[6] Cristina Alcaraz, Gerardo Fernandez, and Fernando Carvajal. Security aspects of scada and dcs environments. In *Critical Infrastructure Protection*, pages 120–149. Springer, 2012.

[7] Tamleek Ali, Mohammad Nauman, Muhammad Amin, and Masoom Alam. Scalable, privacy-preserving remote attestation in and through federated identity management frameworks. In *2010 International Conference on Information Science and Applications*, pages 1–8. IEEE, 2010.

[8] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. μrai: Securing embedded systems with return address integrity. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26*, 2020.

[9] Sami Alsouri, Özgür Dagdelen, and Stefan Katzenbeisser. Group-based attestation: Enhancing privacy and management in remote attestation. In *International Conference on Trust and Trustworthy Computing*, pages 63–77. Springer, 2010.

[10] Sigurd Frej Joel Jørgensen Ankergård, Edlira Dushku, and Nicola Dragoni. State-of-the-art software-based remote attestation: Opportunities and open issues for internet of things. *Sensors*, 21(5):1598, 2021.

[11] Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-taa. In *International conference on security and cryptography for networks*, pages 111–125. Springer, 2006.

[12] Michael Backes, Lucjan Hanzlik, and Jonas Schneider-Bensch. Membership privacy for fully dynamic group signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2181–2198, 2019.

[13] Pranshu Bajpai and Richard Enbody. Memory forensics against ransomware. In *Conference on Cyber Security and Protection of Digital Services*, 2020.

[14] Foteini Baldimtsi and Anna Lysyanskaya. Anonymous credentials light. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1087–1098, 2013.

[15] Alcardo Alex Barakabitze, Arslan Ahmad, Rashid Mijumbi, and Andrew Hines. 5g network slicing using sdn and nfv: A survey of taxonomy, architectures and future challenges. *Computer Networks*, 167:106984, 2020.

[16] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *International conference on the theory and applications of cryptographic techniques*, pages 480–494. Springer, 1997.

[17] Daniel Beer. mspdebug. Retrieved December 20, 2022 from https://github.com/dlbeer/mspdebug.

[18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

[19] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, pages 90–108. Springer, 2013.

[20] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct {Non-Interactive} zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014.

[21] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: efficient, succinct, modular. In *International Conference on Financial Cryptography and Data Security*, pages 393–414. Springer, 2021.

[22] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[23] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 329–349. 2019.

[24] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the sdh assumption in bilinear groups. *Journal of cryptology*, 21(2):149–177, 2008.

[25] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *Annual international cryptology conference*, pages 41–55. Springer, 2004.

[26] Christian Borgelt. Seqwog. Retrieved December 20, 2022 from https://borgelt.net/seqwog.html.

[27] Sean Bowe, Ariel Gabizon, and Matthew D Green. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. In *International Conference on Financial Cryptography and Data Security*, pages 64–77. Springer, 2018.

[28] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the twenty-fourth ACM Symposium on Operating Systems Principles*, pages 341–357, 2013.

[29] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, 2004.

[30] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One tpm to bind them all: Fixing tpm 2.0 for provably secure anonymous attestation. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 901–920. IEEE, 2017.

[31] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In *International Conference on Trust and Trustworthy Computing*, pages 1–20. Springer, 2016.

[32] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: definitions and practical constructions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 262–288. Springer, 2015.

[33] Jan Camenisch and Thomas Groß. Efficient attributes for anonymous credentials. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 345–356, 2008.

[34] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Annual international cryptology conference*, pages 56–72. Springer, 2004.

[35] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. *Technical Report/ETH Zurich, Department of Computer Science*, 260, 1997.

[36] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 21–30, 2002.

[37] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2075–2092, 2019.

[38] Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. Exposure-resilient functions and all-or-nothing transforms. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 453–469. Springer, 2000.

[39] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. Reconciling remote attestation and safety-critical operation on simple iot devices. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[40] Chin-Yao Chang, Richard Macwan, and Sinnott Murphy. Zero-knowledge proof-based approach for verifying the computational integrity of power grid controls. *arXiv preprint arXiv:2211.06724*, 2022.

[41] Roderick Chapman. Sanitizing sensitive data: How to get it right (or at least less wrong…). In *Ada-Europe International Conference on Reliable Software Technologies*, pages 37–52. Springer, 2017.

[42] Melissa Chase, Esha Ghosh, Srinath Setty, and Daniel Buchner. Zero-knowledge credentials with deferred revocation checks. Retrieved Sep 27, 2022 from https://github.com/decentralized-identity/snark-credentials/blob/master/whitepaper.pdf/.

[43] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[44] David Chaum and Eugène van Heyst. Group signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 257–265. Springer, 1991.

[45] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. Property-based attestation without a trusted third party. In *International Conference on Information Security*, pages 31–46. Springer, 2008.

[46] Liqun Chen and Rainer Urian. Daa-a: Direct anonymous attestation with attributes. In *International Conference on Trust and Trustworthy Computing*, pages 228–245. Springer, 2015.

[47] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[48] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an aes implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[49] MoteIV Coorperation. Ultra low power ieee 802.15.4 compliant wireless sensor module. *Datasheet*, 2006.

[50] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[51] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Annual International Cryptology Conference*, pages 174–187. Springer, 1994.

[52] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 45–64. Springer, 2002.

[53] CyBOK. The cyber security body of knowledge - version 1.1.0, 2021. Retrieved December 22, 2022 from https://www.cybok.org/media/downloads/CyBOK_v1.1.0.pdf.

[54] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[55] Heini Bergsson Debes. Code for Key Exiltration attack, 2022. https://github.com/HeiniDebes/Key-Exfil.

[56] Heini Bergsson Debes. Code for ZEKRA, 2022. https://github.com/HeiniDebes/ZEKRA.

[57] Heini Bergsson Debes. Code for ZEKRO, 2022. https://github.com/HeiniDebes/ZEKRO.

[58] Heini Bergsson Debes. Code for RETRACT, 2023. https://github.com/HeiniDebes/RETRACT.

[59] Heini Bergsson Debes and Thanassis Giannetsos. Segregating keys from noncense: Timely exfil of ephemeral keys from embedded systems. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 92–101. IEEE, 2021.

[60] Heini Bergsson Debes and Thanassis Giannetsos. Zekro: Zero-knowledge proof of integrity conformance. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.

[61] Xudong Deng, Chengliang Tian, Fei Chen, and Hequn Xian. Designated-verifier anonymous credential for identity management in decentralized systems. *Mobile Information Systems*, 2021, 2021.

[62] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[63] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[64] Dominic Deuber, Matteo Maffei, Giulio Malavolta, Max Rabkin, Dominique Schröder, and Mark Simkin. Functional credentials. *Proc. Priv. Enhancing Technol.*, 2018(2):64–84, 2018.

[65] DIF. Dif presentation exchange. Retrieved Sep 27, 2022 from https://identity.foundation/presentation-exchange/.

[66] dock. Dock: Verifiable credentials company. Retrieved November 13, 2022 from https://www.dock.io/.

[67] Jacob Eberhardt and Stefan Tai. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091. IEEE, 2018.

[68] Embench. Modern embedded benchmark suite. Retrieved November 13, 2022 from https://www.embench.org/.

[69] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[70] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pages 67–72. IEEE, 1997.

[71] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.

[72] Philippe Fournier-Viger, Cheng-Wei Wu, Antonio Gomariz, and Vincent S Tseng. Vmsp: Efficient vertical mining of maximal sequential patterns. In *Canadian conference on artificial intelligence*, pages 83–94. Springer, 2014.

[73] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[74] Mike Frantzen and Mike Shuey. {StackGhost}: Hardware facilitated stack protection. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[75] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*, 52(4):585–598, 2017.

[76] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. Physical key extraction attacks on pcs. *Communications of the ACM*, 59(6):70–79, 2016.

[77] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[78] Thanassis Giannetsos, Tassos Dimitriou, Ioannis Krontiris, and Neeli R Prasad. Arbitrary code injection through self-propagating worms in von neumann architecture devices. *The Computer Journal*, 53(10), 2010.

[79] Jairo Giraldo, Esha Sarkar, Alvaro A Cardenas, Michail Maniatakos, and Murat Kantarcioglu. Security and privacy in cyber-physical systems: A survey of surveys. *IEEE Design & Test*, 34(4):7–17, 2017.

[80] Ken Goldman. Ibm's software tpm and tss. Retrieved February 24, 2022 from https://sourceforge.net/projects/ibmswtpm2 and sourceforge.net/projects/ibmtpm20tss.

[81] Shafi Goldwasser et al. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[82] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535, 2021.

[83] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 321–340. Springer, 2010.

[84] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326. Springer, 2016.

[85] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 173–184, 2017.

[86] Utku Gulen, Abdelrahman Alkhodary, and Selcuk Baktir. Implementing rsa for wireless sensor nodes. *Sensors*, 19(13):2864, 2019.

[87] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[88] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[89] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. Anonymous single-sign-on for n designated services with traceability. In *European Symposium on Research in Computer Security*, pages 470–490. Springer, 2018.

[90] Lucjan Hanzlik and Daniel Slamanig. With a little help from my friends: constructing practical anonymous credentials. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2004–2023, 2021.

[91] Christopher Hargreaves and Howard Chivers. Recovery of encryption keys from memory using a linear scan. In *2008 Third International Conference on Availability, Reliability and Security*, pages 1369–1376. IEEE, 2008.

[92] Seyed Mahmood Hejazi, Chamseddine Talhi, and Mourad Debbabi. Extraction of forensically sensitive information from windows physical memory. *digital investigation*, 6:S121–S131, 2009.

[93] Caleb Helbling. Directed graph hashing. *arXiv preprint arXiv:2002.06653*, 2020.

[94] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[95] Jianxing Hu, Dongdong Huo, Meilin Wang, Yazhe Wang, Yan Zhang, and Yu Li. A probability prediction based mutable control-flow attestation scheme on embedded platforms. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 530–537. IEEE, 2019.

[96] iden3. iden3. Retrieved November 13, 2022 from https://github.com/iden3.

[97] iden3. iden3/circom: zksnark circuit compiler. Retrieved November 13, 2022 from https://github.com/iden3/circom.

[98] iden3. iden3/snarkjs: zksnark implementation in javascript & wasm. Retrieved November 13, 2022 from https://github.com/iden3/snarkjs.

[99] T. Instruments. 2.4 ghz ieee 802.15.4/zigbee-ready rf transceiver, 2011.

[100] Texas Instruments. AES-128. Retrieved December 20, 2022 from https://ti.com/tool/AES-128.

[101] Texas Instruments. Msp430x1xx family user's guide (rev. f). 2006.

[102] Texas Instruments. Msp430 embedded application binary interface, 2013.

[103] Trent Jaeger, Reiner Sailer, and Umesh Shankar. Prima: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28, 2006.

[104] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 143–154. Springer, 1996.

[105] Brian Kaplan et al. Ram is key extracting disk encryption keys from volatile memory. 2007.

[106] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732, 1992.

[107] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. In *NDSS*, 2021.

[108] Stephan Kleber, Florian Unterstein, Matthias Matousek, Frank Kargl, Frank Slomka, and Matthias Hiller. Secure execution architecture based on puf-driven instruction level code encryption. *Cryptology ePrint Archive*, 2015.

[109] Tobias Klein. All your private keys are belong to us. Technical report, 2006.

[110] Tommy Koens, Coen Ramaekers, and Cees Van Wijk. Efficient zero-knowledge range proofs in ethereum. *ING, blockchain@ ing. com*, 2018.

[111] kokke. Tiny AES. Retrieved December 20, 2022 from https://github.com/kokke/tiny-AES-c.

[112] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and Other Botnets. *Computer*, 50(7):80–84, 2017.

[113] Ahmed Kosba. Java zksnark library. Retrieved December 20, 2022 from https://github.com/akosba/jsnark.

[114] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961. IEEE, 2018.

[115] Nikos Koutroumpouchos, Christoforos Ntantogian, Sofia-Anna Menesidou, Kaitai Liang, Panagiotis Gouvas, Christos Xenakis, and Thanassis Giannetsos. Secure edge computing with lightweight control-flow property-based attestation. *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 84–92, 2019.

[116] Hugo Krawczyk and Tal Rabin. Chameleon hashing and signatures. 1998.

[117] Boyu Kuang, Anmin Fu, Lu Zhou, Willy Susilo, and Yuqing Zhang. Do-ra: data-oriented runtime attestation for iot devices. *Computers & Security*, 97:101945, 2020.

[118] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[119] SCIPR Lab. C++ zksnark library. Retrieved December 20, 2022 from https://github.com/scipr-lab/libsnark.

[120] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security Privacy*, 9(3):49–51, 2011.

[121] Benjamin Larsen, Heini Bergsson Debes, and Thanassis Giannetsos. Cloudvaults: Integrating trust extensions into system integrity verification for cloud-based environments. In *European Symposium on Research in Computer Security*, pages 197–220. Springer, 2020.

[122] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

[123] Panagiotis Liakos, Katia Papakonstantinopoulou, and Alex Delis. Realizing memory-optimized distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(4):743–756, 2017.

[124] Yan Lin, Xiaoyang Cheng, and Debin Gao. Control-flow carrying code. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 3–14, 2019.

[125] Jingbin Liu, Qin Yu, Wei Liu, Shijun Zhao, Dengguo Feng, and Weifeng Luo. Log-based control flow attestation for embedded devices. In *International Symposium on Cyberspace Safety and Security*, pages 117–132. Springer, 2019.

[126] Wu Luo, Wei Liu, Yang Luo, Anbang Ruan, Qingni Shen, and Zhonghai Wu. Partial attestation: towards cost-effective and privacy-preserving remote attestations. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 152–159. IEEE, 2016.

[127] Wu Luo, Qingni Shen, Yutang Xia, and Zhonghai Wu. {Container-IMA}: A privacy-preserving integrity measurement architecture for containers. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 487–500, 2019.

[128] John Lyle and Andrew Martin. On the feasibility of remote attestation for web services. In *2009 International Conference on Computational Science and Engineering*, volume 3, pages 283–288. IEEE, 2009.

[129] Roel Maes and Ingrid Verbauwhede. Physically unclonable functions: A study on the state of the art and future research directions. In *Towards Hardware-Intrinsic Security*, pages 3–37. Springer, 2010.

[130] matteocam. libsnark: a c++ library for zksnark proofs. Retrieved November 13, 2022 from https://github.com/matteocam/libsnark-lego.

[131] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[132] Silvio Micali. Cs proofs. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 436–453. IEEE, 1994.

[133] Microsoft. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, 2019. Retrieved December 22, 2022 from https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL.

[134] László Monostori, Botond Kádár, Thomas Bauernhansl, Shinsuke Kondoh, Soundar Kumara, Gunther Reinhart, Olaf Sauer, Gunther Schuh, Wilfried Sihn, and Kenichi Ueda. Cyber-physical systems in manufacturing. *Cirp Annals*, 65(2):621–641, 2016.

[135] Tilo Müller, Andreas Dewald, and Felix C Freiling. Aesse: a cold-boot resistant implementation of aes. In *Proceedings of the Third European Workshop on System Security*, pages 42–47, 2010.

[136] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, volume 17, 2011.

[137] Assa Naveh and Eran Tromer. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 255–271. IEEE, 2016.

[138] NSA. Software memory safety - cybersecurity information sheet, 2022. Retrieved December 22, 2022 from https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.

[139] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. {APEX}: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 771–788, 2020.

[140] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: A minimalistic approach for control-flow attestation using verified proofs of execution. *arXiv preprint arXiv:2011.07400*, 2020.

[141] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Dialed: Data integrity attestation for low-end embedded devices. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 313–318. IEEE, 2021.

[142] NVD. CVE-2020-9395 Detail, 2020.

[143] NVD. CVE-2021-44228, 2021.

[144] Alex Ozdemir, Riad S Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2075–2092, 2020.

[145] Christian Paquin and Greg Zaverucha. U-prove cryptographic specification v1. 1. *Technical Report, Microsoft Corporation*, 2011.

[146] T Paul Parker and Shouhuai Xu. A method for safekeeping cryptographic keys from memory disclosure attacks. In *International Conference on Trusted Systems*, pages 39–59. Springer, 2009.

[147] Bryan Parno et al. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[148] Andrew Paverd, Andrew Martin, and Ian Brown. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Tech. Rep*, 2014.

[149] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.

[150] Torbjörn Pettersson. Cryptographic key recovery from linux memory dumps. *Chaos Communication Camp*, 2007, 2007.

[151] Lukas Petzi, Ala Eddine Ben Yahya, Alexandra Dmitrienko, Gene Tsudik, Thomas Prantl, and Samuel Kounev. Scraps: Scalable collective remote attestation for pub-sub iot networks with untrusted proxy verifier. In *USENIX Security 22*, 2022.

[152] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. *Cryptology ePrint Archive*, 2022.

[153] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, 2004.

[154] Shahrokh Saeednia, Steve Kremer, and Olivier Markowitch. An efficient strong designated verifier signature scheme. In *International conference on information security and cryptology*, pages 40–54. Springer, 2003.

[155] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security symposium*, volume 13, pages 223–238, 2004.

[156] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. {Policy-Sealed} data: A new abstraction for building trusted cloud services. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 175–188, 2012.

[157] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*, pages 459–474. IEEE, 2014.

[158] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[159] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *International Conference on Applied Cryptography and Network Security*, pages 346–366. Springer, 2016.

[160] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.

[161] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. Taking {Proof-Based} verified computation a few steps closer to practicality. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 253–268, 2012.

[162] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[163] Adi Shamir and Nicko Van Someren. Playing "hide and seek" with stored keys. In *International conference on financial cryptography*, 1999.

[164] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[165] Johannes Sianipar, Muhammad Sukmana, and Christoph Meinel. Moving sensitive data against live memory dumping, spectre and meltdown attacks. In *2018 26th International Conference on Systems Engineering (ICSEng)*, 2018.

[166] Patrick Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 73–82, 2011.

[167] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018.

[168] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[169] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[170] Syh-Yuan Tan and Thomas Groß. Monipoly—an expressive q-sdh-based anonymous attribute-based credential system. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 498–526. Springer, 2020.

[171] Benjamin Taubmann, Omar Alabduljaleel, and Hans P Reiser. Droidkex: Fast extraction of ephemeral tls keys from the memory of android apps. *Digital Investigation*, 26:S67–S76, 2018.

[172] TCG. Tpm 2.0 library - trusted computing group. Retrieved February 24, 2022 from https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[173] TCG. *TCG Guidance for Securing Network Equipment*, Jan 2018.

[174] TCG. *TPM 2.0 Keys for Device Identity and Attestation*, Jan 2018.

[175] TinyOS. Tinyos. Retrieved December 20, 2022 from https://github.com/tinyos/tinyos-main.

[176] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. {ScaRR}: Scalable runtime remote attestation for complex systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 121–134, 2019.

[177] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 19–34. IEEE, 2017.

[178] trinsic. Trinsic - a full-stack self-sovereign identity (ssi) platform. Retrieved November 13, 2022 from https://trinsic.id/.

[179] W3C. Verifiable credentials data model v1.1. Retrieved Sep 27, 2022 from https://www.w3.org/TR/vc-data-model/.

[180] Riad S Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. *Cryptology ePrint Archive*, 2014.

[181] K Watsen, I Farrer, and M Abrahamsson. Secure zero touch provisioning (sztp). *Internet Requests for Comments, RFC Editor, IETF, Wilmington, DE, USA, Tech. Rep*, 8572, 2019.

[182] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.

[183] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[184] Siyuan Xin, Yong Zhao, and Yu Li. Property-based remote attestation oriented to cloud computing. In *2011 Seventh International Conference on Computational Intelligence and Security*, pages 1028–1032. IEEE, 2011.

[185] Sachiko Yoshihama, Tim Ebringer, Megumi Nakamura, Seiji Munetoh, and Hiroshi Maruyama. Ws-attestation: Efficient and fine-grained remote attestation on web services. In *IEEE International Conference on Web Services (ICWS'05)*. IEEE, 2005.

[186] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391. IEEE, 2017.

[187] Yumei Zhang et al. Recfa: Resilient control-flow attestation. In *Annual Computer Security Applications Conference*, ACSAC '21, 2021.

[188] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1219–1236, 2020.

Appendix A

Malcode Implementation Details

This appendix describes the methotolody for realizing the building blocks in Section 2.6.2, i.e., establishing the necessary callbacks and timers on the target device (described in Section 2.4.1).

A.1 Challenges

Since ISRs are programmed together with program code into flash, we must reprogram the flash segment containing the timer and RX ISRs (Section 2.6.1) to achieve the necessary callbacks. Fortunately, MSP430 accommodates in-system programmable flash memory [101], as is typical for sensor motes to enable over-the-air field upgrades. Thus, with the advantageous Von Neumann memory model and lack of no-execute (NX), an adversary $\mathcal{A}$ can easily reprogram flash from RAM.

A.1.1 Administration of Periodic Invocation

The MSP430-F1611 MCU has built-in MMIO configurations for two flexible 16-bit (asynchronous) timers, named Timer A (TA) and Timer B (TB), respectively. Each timer has multiple capture-compare units (CCUs), one control register, T*x*CTL, for configuring the timer, e.g., specifying a mode of operation and clock source, and is coupled with one 16-bit counter register, T*x*R, which increments or decrements (depending on the mode of operation) with each rising edge of the selected clock signal [101]. Each CCU has one capture/compare register (CCR), one capture/compare control register (CCTL) and operates in either capture or compare mode, as determined by the CCTL register. In compare mode, the value to be compared to is first loaded into the T*x*CCR, and when T*x*R equals that value, it raises the capture/compare interrupt flag (CCIFG) for that T*x*CCR. Thus, to establish periodic interrupts, we configure TBCTL in *up* mode and say, TBCCR6, in compare mode, such that TBCCR6's CCIFG gets raised whenever TBR equals whichever value we store in TBCCR0 (the period). To react to this interrupt, we must reprogram the appropriate ISR, which for TBCCR6 is the IVTBL entry at FFF8h (used for TBCCR1

to TBCCR6). However, since multiple TBCCR CCIFGs are merged into this ISR, we must consult the timer interrupt vector register (TBIV) stored at 011Eh to determine whether TBCCR6 caused the interrupt. Finally, as the clock source (to regulate the period between stackshots), we use the auxiliary clock (ACLK), sourced from a 32 kHz watch crystal to achieve a granularity of 32 Cycles Per Millisecond (CPMS).

A.1.2 Narrowing the Time Before the KEW

To position the stack acquisition closer to the reception handler's invocation, we can follow the documented behavior of the CC2420. The MoteIV Tmote Sky module [49] has a CC2420 [99] transceiver, which the MCU controls using an SPI link managed by UART0 and a series of I/O lines and interrupts. Specifically, MSP430 devices have up to 6 digital I/O ports, numbered P1 to P6. Each port has eight I/O pins, numbered Px0 to Px7, and four MMIO registers: PxSEL, PxDIR, PxOUT, and PxIN. When the CC2420 interrupts the CPU about an incoming packet, the packet is incrementally read from the CC2420 reception queue (RXFIFO) into the MCU's U0RXBUF memory buffer (located at 76h). On successive reads, the USART0 RX ISR (IVTBL entry FFF2h) [101] is invoked, and only when the entire packet resides in application memory then the reception handler gets invoked. Fortunately, the transceiver requires that the output of the Chip Select (CSn) pin, which is connected to pin 2 (3rd bit) of port 4 on the MCU, must remain low while there is any communication with it (read or write). Therefore, by polling the CSn (bit 3 in the P4OUT register located at 01Dh) and noticing a transition from low (0) to high (1), then we are sure that the invocation of the reception handler is imminent.

A.1.3 Programming Flash

Flash can be programmed using a JTAG interface, the Bootstrap loader (BSL), or using a *custom solution* through user developed software. Each method has some method of protection. The JTAG is protected by a fuse. Blowing the fuse completely disables the JTAG port and is irreversible. Access to the MSP430 flash memory via the BSL is protected by a 256-bit, user-defined password. But the flexibility of in-system programmable flash memory is advantageous to $\mathcal{A}$. Because the memory model is Von Neumann and there is no no-execute (NX) protection, code can easily be executed from RAM. This includes flash read and write operations. The F1611 has three dedicated read/write flash memory control (FCTL) registers, FCTL1, FCTL2, and FCTL3 (located at 0128h, 012Ah, and 012Ch, respectively). The FCTLx registers are 16-bit, "password-protected", read/write registers. Any read or write access must use word instructions and write accesses must include the write password *0A5h* in the upper byte [101]. The key violation flag KEYV is set when any of the flash control registers are written with an incorrect password. When this occurs, a PUC is generated immediately resetting the device. Furthermore, it is also advised to disable the *watchdog timer* (WDT) through the WDT control (WDTCTL) register before writing to flash. The main task of the WDT module is to perform a controlled system restart after a software problem occurs (obviously not desirable). As with the FCTL registers, the WDTCTL register is password protected and requires the password *05Ah* in the upper byte when reading or writing to it. Note, because this password is known to $\mathcal{A}$ it is *trivial* to reprogram flash during run-time. Furthermore, reading flash memory *does not* require a password. The only challenge when writing to flash is to properly

configure the flash controller with clocks and sources. More importantly, when flash operations are conducted with the CPU executing code resident in RAM the CPU is not automatically held while the operation is being performed. Therefore, it is required that the code (malcode) includes logic to poll the BUSY (FCTL3 register) flag. When FCTL3 transitions from 1 (busy) to 0 (idle) we know that we can access flash addresses again.

Note, on the MSP430, flash must first be erased (each bit set to 1) before a value can be written (individual bits changed to 0). However, the MSP430 only supports flash erasure at a segment granularity (see Figure 2.2 for the segmentation). Furthermore, identical to write operations, when initiated from RAM the flash controller is skipped and the BUSY flag must be polled for the end of the erase cycle. Some MSP430 MCUs allow block write to accelerate the process when many sequential bytes or words need to be programmed.

A.2 Crafting the malcode

We proceed with a brief description of our devised and application-agnostic malcode, which to recap, leverages TinyOS's minimalism, the run-time programmability of the MCU, and incorporates the methodology in Appendix A.1.

The malcode is a collection of five code segments and, for brevity, accepts 18 configurable parameters, which are presented together with arguments used during our experiments (Section 2.7) in Table A.1. Besides the predefined $\mathcal{WD}$ (see Malcode 3) and $\mathcal{FE}$ (Malcode 4), the malcode comprises a *Setup Engine* ($\mathcal{SE}$, Malcode 1) and an *ISR Injector* ($\mathcal{ISRI}$, Malcode 2), where $\mathcal{SE}$ is the initial *triggering* of the malcode (Section 2.6.1). When invoked, $\mathcal{SE}$ uses $\mathcal{ISRI}$ to inject callbacks to $\mathcal{WD}$ and $\mathcal{FE}$ into the RX and timer ISRs. Note that for brevity, we omitted the final segment, which transmits stackshots to $\mathcal{A}$. Nonetheless, to maintain state and to reprogram flash, the malcode uses unoccupied RAM space, which, because MSP430 requires flash programming on a segment granularity, must be ≥ 512 bytes, such that $\mathcal{ISRI}$ can temporarily copy, reprogram, and write back segments.

Note that during the experiments in Section 2.7 we set $\text{PARAM}_{\text{period}} = \lceil \text{CW/CF} * \text{CPMS} \rceil$, where CPMS (Cycles Per Millisecond) is 32. Thus, for CW = 97.57 ms, the time period at CF 128 will be 0.76 ms and $\text{PARAM}_{\text{period}} = \lceil 24.39 \rceil = 25$.

A.2.1 Methodical Execution

Once the $\mathcal{A}$ sends an activator packet which tricks the CPU's Program Counter (PC) register to point to the beginning of $\mathcal{SE}$ (see Fig. 2.4), $\mathcal{ISRI}$ is used to inject a callback to $\mathcal{WD}$ in USART0's RX ISR and another to $\mathcal{FE}$ in TB's ISR (Appendix A.1). On each invocation, $\mathcal{ISRI}$: (i) identifies in which flash segment the target ISR is located, (ii) copies that segment into RAM (where it can manipulate it freely), (iii) overwrites the first two push statements in the ISR's prologue with a branch (jump) to the appropriate malcode component, (iv) clears the segment in flash memory, and finally (v) writes the manipulated segment back into its original slot. Thus, when either ISR triggers, the appropriate malcode is invoked. Subsequently, $\mathcal{SE}$ gracefully resumes the reception handler by reverting control-flow to the reception handler's original return address ($\text{ADDR}_{\text{restore}}$, see Table A.1).

Table A.1: Malcode parameters and demonstrative arguments.

Parameter	Argument	Description
Config		
$\text{PARAM}_{\text{usartISR}}$	FFF2h	Target USART0RX IVTBL entry
$\text{PARAM}_{\text{timerISR}}$	FFF8h	Target TB IVTBL entry
$\text{PARAM}_{\text{CCTL}}$	018Eh	Target TBCCTL6
$\text{PARAM}_{\text{CCIFG}}$	0Ch	Value of TBCCR6 CCIFG in TBIV
$\text{PARAM}_{\text{runs}}$	1	# of successive runs (receptions)
$\text{PARAM}_{\text{stackshots}}$	CF	# of stackshots in each run
$\text{PARAM}_{\text{period}}$	$\lceil \text{CW}/\text{CF} * \text{CPMS} \rceil$	Time between stackshots
PARAM_{rg}	64	# of bytes to extract
States (updated during run-time)		
ADDR_{run}	2000h	Current run count
$\text{ADDR}_{\text{stackshot}}$	2001h	Current stackshot count
$\text{ADDR}_{\text{tmpPtr}}$	2002h	Current offset in temporary storage
ADDR_{tmp}	2200h	Temporary storage ≥ 512 bytes (see Fig. 2.5)
$\text{ADDR}_{\text{restore}}$	*	Restoration memory address (Fig. 2.4)
Demonstrative memory placement of the malcode		
$\text{ADDR}_{\mathcal{SE}}$	2004h	Start address of Malcode 1 (Fig. A.1)
$\text{ADDR}_{\mathcal{ISRI}}$	202Ch	Start address of Malcode 2 (Fig. A.3)
$\text{ADDR}_{\mathcal{WD}}$	20C4h	Start address of Malcode 3 (Fig. A.2)
$\text{ADDR}_{\mathcal{FE}}$	20FCh	Start address of Malcode 4 (Fig. A.4)

At this stage, the malcode is armed but lies dormant as it awaits reception. Upon reception, the $\mathcal{WD}$ awakens, and unless ADDR_{run} has reached the $\text{PARAM}_{\text{runs}}$ threshold, TB is started to periodically (regulated with $\text{PARAM}_{\text{period}}$) invoke $\mathcal{FE}$. Since different timers can run contemporaneously and $\mathcal{FE}$'s callback resides in TB's ISR's prologue, $\mathcal{FE}$ consults TBIV to determine whether the CCIFG of the targeted CCU is raised (i.e., since we consider CCU6, whether TBIV has the value 0Ch [101]). If so, $\mathcal{FE}$ determines whether the CSn has become high (Appendix A.1), and if it has, advances to copy PARAM_{rg} bytes from where the SP currently points (excluding the first five words emitted by the interrupt and the $\mathcal{FE}$'s prologue) into ADDR_{tmp}, using $\text{ADDR}_{\text{tmpPtr}}$ as an offset, and updates $\text{ADDR}_{\text{stackshot}}$ and $\text{ADDR}_{\text{tmpPtr}}$ accordingly. Once the $\text{PARAM}_{\text{stackshots}}$ threshold is reached, $\mathcal{FE}$ stops its timer, increments ADDR_{run}, and transmits the accumulated stackshots to $\mathcal{A}$. Finally, $\mathcal{FE}$ resumes the TB's ISR.

A.2.2 Assembly

The assembly code for the malcode components, $\mathcal{SE}$, $\mathcal{ISRI}$, $\mathcal{WD}$, and $\mathcal{FE}$ (omitting the fifth component for transmitting stackshots), are presented in Fig. A.1, Fig. A.3, Fig A.2, and Fig A.4, respectively, and take Table A.1 as input. Note, before reprogramming flash, $\mathcal{ISRI}$ first disables the *watchdog timer* (WDT) module to halt controlled system restarts. Further, because $\mathcal{SE}$ occurs in the reception task's context (due to the buffer-overflow described in Section 2.5.4), and both $\mathcal{WD}$ and $\mathcal{FE}$ will occur in ISR contexts, registers (states) must be preserved and restored upon returning, regardless of calling conventions, not to disturb the original program's data-flow. However, because ISRs

conventionally preserve the Status Register (SR) and PC, these need not be considered by $\mathcal{WD}$ nor $\mathcal{FE}$.

Malcode 1: Setup Engine $\mathcal{SE}$ (40 bytes)

```
1:  PUSH R15 - R14
2:  MOV.B #0, &ADDR_run
3:  MOV &PARAM_usartISR, R14
4:  MOV #ADDR_WD, R15
5:  CALL #ADDR_ISRI
6:  MOV &PARAM_timerISR, R14
7:  MOV #ADDR_FE, R15
8:  CALL #ADDR_ISRI
9:  POP R14 - R15
10: BR #ADDR_restore
```

Figure A.1: Demonstrative Setup Engine's malcode.

Malcode 3: Watchdog $\mathcal{WD}$ (≥ 54 bytes)

```
1:  PUSH R15 - R14
2:  CMP.B #PARAM_runs, &ADDR_run     / done?
3:  JZ line 12     / if yes, skip
4:  BIT #0x0010, &PARAM_CCTL     / timer started?
5:  JC line 12     / if yes, skip
6:  MOV.B #0, &ADDR_stackshot
7:  MOV #0, &ADDR_tmpPtr
8:  MOV #PARAM_period, &0x0192     / store in TBCCR0
9:  MOV #0x0010, &PARAM_CCTL     / enable interrupts
10: CLR &0x0190     / reset TBR
11: MOV #0x1910, &0x0180     / TBCTL in up mode
12: MOV &PARAM_usartISR, R15
13: ADD #0x0004, R15     / skip branch to self
14: BR R15     / allow ISR to progress
```

Figure A.2: Demonstrative watchdog's malcode.

Malcode 2: ISR Injector $\mathcal{ISRI}$ (152 bytes)

```
1:  PUSH R2, R13 - R10
2:  MOV #0x5A80, &0x0120    / stop WDT module
3:  MOV.B R14, R13          / copy LSB as offset
4:  SWPB R14                / swap MSB and LSB
5:  MOV.B R14, R12          / MSB = segment start
6:  MOV.B R12, R11          / copy MSB for testing
7:  AND.B #0x01, R11        / 1 if MSB is odd
8:  TST.B R11               / test if even or odd
9:  JZ 0x6                  / skip if even
10: DEC.B R12               / else, decrement MSB
11: ADD #0x0100, R13        / make offset odd
12: ADD #ADDR_tmp, R13      / add segment offset
13: SWPB R12
14: MOV R12, R11            / copy segment start
15: ADD #0x0200, R11        / end = start + 512B
16: MOV #ADDR_tmp, R10
17: MOV R12, R14            / copy segment start
18: MOV R14+, 0x0000(R10)
19: INCD R10
20: CMP R14, R11            / end of flash segment?
21: JNZ -0xA                / go back 10 bytes
22: MOV #0x4030, 0x0000(R13)  / swap(PUSH, BR)
23: MOV R15, 0x0002(R13)    / swap(PUSH, callback)
24: MOV #0xA542, &0x012A    / use MCLK/3
25: MOV #0xA502, &0x0128    / set ERASE bit
26: MOV #0xA500, &0x012C    / remove LOCK bit
27: CLR 0x0000(R12)         / erase segment
28: BIT #0x0008, &0x012C    / check write status
29: JZ -0x6                 / loop until done
30: MOV #ADDR_tmp, R10
31: MOV #0xA540, &0x0128    / set WRT bit
32: MOV R10+, 0x0000(R12)   / write word
33: INCD R12
34: BIT #0x0001, &0x012C    / check busy status
35: JNZ -0x6                / loop until not busy
36: CMP R12, R11            / end of flash segment?
37: JNZ -0x10
38: MOV #0xA500, &0x0128    / remove WRT bit
39: MOV #0xA510, &0x012C    / set LOCK bit
40: POP R10 - R13, R2
41: RET
```

Figure A.3: Demonstrative ISR injector's malcode.

Malcode 4: Frame Extractor $\mathcal{FE}$ ($\geq$ 88 bytes)

```
1:  PUSH R15 - R14 and R13
2:  CMP.B #PARAM_CCIFG, &0x011E    / did the target timer fire?
3:  JNZ line 24
4:  BIT.B #0x04, &0x001D    / CSn high?
5:  JNC line 24
6:  MOV #ADDR_tmp, R13
7:  MOV &ADDR_tmpPtr, R13    / continue from tmpPtr offset
8:  MOV R1, R14    / R1 is the SP
9:  MOV R1, R15
10: ADD #0x000A, R14    / ignore 10 bytes (SR, PC, 3xPUSH)
11: ADD #0x000A, R15
12: ADD #PARAM_rg, R15
13: MOV @R14+, 0x0000(R13)
14: INCD R13
15: CMP R14, R15
16: JNZ -0xA
17: INC.B &ADDR_stackshot
18: ADD #PARAM_rg, &ADDR_tmpPtr    / increment offset
19: CLR &0x0190    / reset TBR
20: CMP.B #PARAM_stackshots, &ADDR_stackshot    / done?
21: JNZ 0x8
22: MOV #0, &PARAM_CCTL    / disable interrupts
23: INC.B &ADDR_run
    CC2420 TRANSMIT
24: MOV &PARAM_timerISR, R15
25: ADD #0x0004, R15    / skip branch to self
26: POP R13
27: BR R15    / allow ISR to progress
```

Figure A.4: Demonstrative frame extractor's malcode.