Nonlinear Dyn (2020) 99:3065–3087
https://doi.org/10.1007/s11071-020-05463-3

ORIGINAL PAPER

An authentication protocol based on chaos and zero knowledge proof

Will Major · William J. Buchanan · Jawad Ahmad

Received: 11 July 2019 / Accepted: 1 January 2020 / Published online: 21 January 2020

Abstract Port Knocking is a method for authenticating clients through a closed stance firewall, and authorising their requested actions, enabling severs to offer services to authenticated clients, without opening ports on the firewall. Advances in port knocking have resulted in an increase in complexity in design, preventing port knocking solutions from realising their potential. This paper proposes a novel port knocking solution, named Crucible, which is a secure method of authentication, with high usability and features of stealth, allowing servers and services to remain hidden and protected. Crucible is a stateless solution, only requiring the client memorise a command, the server's IP and a chosen password. The solution is forwarded as a method for protecting servers against attacks ranging from port scans, to zero-day exploitation. To act as a random oracle for both client and server, cryptographic hashes were generated through chaotic systems.

Keywords ZKP · Chaos hash · Port knocking · Random beacons · Attack model

1 Introduction

Port knocking, if integrated into a security environment, can offer an additional layer of authentication for servers, furthering a defence-in-depth approach, and can conceal the presence of services. It is suited to defending against attacks directed at servers, ranging from automatic scanning, as part of attack-chain reconnaissance, to precisely targeted zero-day exploitation. Port knocking solutions have progressed and changed dramatically, since their conception as a simple tool for opening firewall ports. Many modern port knocking implementations have accumulated layers of complexity in the process of removing specific vulnerabilities from their predecessors. This complexity issue is further exacerbated by components in port knocking that are mutually incompatible: replay protection can result in desynchronisation problems, and interactive authentication requires the server to forgo its 'silent' role. To combat this, many modern port knocking implementations in academic literature take an ad-hoc approach to fixing existing vulnerabilities without considering a holistic viewpoint, nor the minimalist lineage upon which port knocking was founded.

Port knocking can be considered a stealthy method of authentication and command execution, allowing a covert channel to exist between a client and server, across an untrusted network such as the Internet. When implemented properly, port knocking should be difficult to discover through passive surveillance of network traffic, or active reconnaissance of the server. Port knocking allows a server to conceal not only its individual services, but also its *role* as a server. Numbers Stations are a Cold War era covert channel using radio broadcasts of spoken number values (amongst

W. Major · W. J. Buchanan (✉)· J. Ahmad
Blockpass ID Lab, Edinburgh Napier University, Edinburgh, UK
e-mail: w.buchanan@napier.ac.uk

other methods), suspected to communicate with intelligence assets in the field [1]. In modern day computing, Meltdown [2] and Spectre [3] are examples of serious vulnerabilities enabling covert channels for exfiltrating data from a victim's machine.

Port knocking can simply be described as a means for communicating with a machine that is protected by a closed stance firewall. This entails a client machine, operated by a user, sending a message—or *knock*—to a firewall. The firewall in this instance will be referred to as the *server*, for its role in receiving the knock, and the subsequently authorised actions it performs. This terminology further aims to clarify in situations involving multiple intermediaries (such as *additional* firewalls) between client and server. The networks across which this transaction can take place could be local, or, as is the common use case, remote. Port knocking is by design aimed at operating over untrusted networks, such as the Internet, wherein malicious actors of varying capabilities aim to subvert security measures. Port knocking methods generally use cryptographic primitives such as hash and encryption functions. In the proposed port knocking method, we have utilised the Chirkov standard map and absolute chaotic map for the hash that ensures the output knock is different for each session. Additionally, the chaos-based hash function provides a lightweight secure solution and as a result, the proposed scheme will not require third-party libraries. Moreover, chaos-based port knocking scheme will have a number of properties such as sensitivity to initial conditions, non-periodicity, ergodicity, and attack complexity which strengthen the port knocking scheme.

This paper offers three novel port knocking prototypes: zero knowledge proofs and chaos-based cryptography; a combination of chaos-based cryptography and random beacons; and 'Crucible' which is combines random beacons and password-based key derivation. Replay protection and NAT compatibility are considered significant issues in port knocking technology. This paper proposes novel approaches for dealing with these problems.

The rest of the paper is organised as follows. Related work is discussed in Sect. 2. Section 3 discusses design and the proposed methodology. Experimental results are presented in Sect. 4. The proposed method is evaluated in Sect. 5. Research findings are concluded in Sect. 6.

2 Related work

Port knocking allows authentication to a host without requiring open ports, and without requiring modification of the underlying protocol [4]. Furthermore, the presence of port knocking can be difficult to detect by sniffing traffic, and almost impossible to detect by probing a server [5]. By making services 'invisible', a machine's function as a *server* can be hidden from an attacker [6], offering a level of anonymity. Port knocking disrupts attacker reconnaissance by denying fingerprinting efforts against open ports. This prevents automated and manual scanning efforts, and reduces "malicious information gathering capabilities" [7]. In the same vein, vulnerability scanning and discovery are impeded, and thus port knocking can be considered one of the few non-reactive defences against zero-day attacks [8]. In this manner, port knocking can provide protection for legacy and proprietary services with "insufficient integrated security", or for services with "known unpatched vulnerabilities" [9]. Port knocking also provides an additional layer of security and authentication, that any malicious actors must overcome before attacking the hidden service itself [7,9].

Port knocking can incur load and performance loss on networks and systems [5], the latter applying an overhead for each connection [4]. In addition, a number of ports may need to be "allocated for exclusive use by port knocking" [10]. Port knocking implementations may require user training [4] and client systems need to implement port knocking, which may require maintenance of dedicated remote client software [7]. Authentication in port knocking is largely facilitated through pre-shared keys (or other secrets), meaning secure key distribution and management become requirements. Port knocking also adds an additional layer of complexity into the protection of assets [7], and another attack vector to defend against. The fail-closed stance of port knocking may result in inaccessibility of services, should the authentication mechanism fail on the server [4,10]. Such a failure in this mechanism could render the server "unreachable or more easily compromised" [7].

2.1 Mechanics and architecture

One of the main differentials when considering port knocking solutions is the number of packets that are required to authenticate the client with the server. Traditional port knocking solutions sent a series of packets (each *knock*) across the Internet, where the *authenticating data* would be communicated via this collection of knocks, known as a *knock sequence*. More recently, a single packet has been used to wholly and atomically send authenticating data to the server, in a single knock, giving rise to the term “Single Packet Authorisation” [11]. Srivastava et al. [5] raises an immediate problem with using multiple packets to authenticate, in that network and routing issues between the server and the client, such as latency (from congestion [7]) or packet drops, will cause the authenticating packets to arrive out-of-order. If the knock sequence doesn’t match the server’s expectations, the client likely won’t be authenticated. Sel [12] offers a solution to this problem by including a sequence number within each knock’s authenticating data, allowing the knocks to be reassembled after they are received.

2.2 Non-interactive or interactive server

While traditionally communications in a port knocking implementation would be limited to unidirectional client to server messaging, some modern variations conversely include server to client communication. This could also be described as unilateral or bilateral communication. Sel [13] for example, use client server conversations to negotiate a session key, which is then used to authenticate application data *after* port knocking has concluded. Tiwari [14] includes a lengthy discussion on this topic. Incorporating challenge and response mechanisms into port knocking can be seen to enable *fresh* authentication, whereby random challenges are used to prove the identity of a peer. Sel [12] also reiterates the advantages of interaction in providing freshness. This approach avoids the drawbacks of other solutions for replay protection, such as requiring time or state-based synchronisation, as is discussed further in Sect. 2.4. Al-Bahadili and Hadi [15] implement such a solution, where as proof of identity, the client is sent an encrypted random number, which it must decrypt and return, to prove it possesses the associated pre-shared key, thus authenticating the client. The authors describe this method as mutual authentication, though it is unclear exactly how the server is authenticated to the client.

2.3 Multi-party and multi-channel involvement

More recently, port knocking solutions have been forwarded that eschew the traditional client-server dynamic, and opt to include additional parties in the protocol. Srivastava et al. [5] sends the client cryptographic information, including a one time key, and a random number, via an out of band, dedicated SMS channel. The random number is used to generate a client-spoofed IP, for preventing client identification by a listening attacker. The one time key is used to encrypt the client’s authenticating data, which is then sent as a port knock to the server. As each knock attempted in this way is unique, protection against replay attacks is provided, with the added bonus of making the port knocking traffic difficult for an attacker to identify. Liew et al. [16] use a similar approach, where SMS is instead used to deliver information from which IPSec tunnel keys are derived, and used to setup a secure channel between the client and server. Popeea et al. [4] enforce time synchronisation between client and server by having them issue NTP requests ahead of a knock, and on startup, respectively. Maintaining accurate timing allows the time-based authentication to be more granular, reducing the window of opportunity for replay attacks.

2.4 Replay protection

A port knock authenticates the client to a server, typically by proving the client possesses a secret, such as a key or password. In proving this over the wire, encryption or hashing are employed to prevent eavesdroppers from learning the secret. An adversary listening to communications between the knocking client and listening server could record a knock and *replay* it back to the server in order to repeat its intended effect. Such attacks have been cited as the main vulnerability affecting basic port knocking implementations [7]. These attacks are prevented by ensuring that each time a port knock is executed, the authenticating data are unique, or *fresh*, and further ensuring that an attacker is unable to generate a fresh knock. Many of the techniques used to achieve freshness carry similarities with one time password (OTP) protocols [17].

3 Design and methodology

3.1 Zero knowledge

A zero knowledge proof (ZKP) is a cryptographic protocol allowing one to prove they posses information to a verifying party, without revealing any of the underlying information itself [18]. Alternatively defined by [19], a ZKP shows “a statement to be true without revealing anything other than the veracity of the statement to be proven”.

As a real-world example (modified from [20]), to prove someone could distinguish between two different types of wine, a number of blind-tests could be performed until the claim was proven or refuted. If the claimant identified the wine correctly each time, the verifier can be reasonably sure of the claimant’s distinguishing ability, without themselves learning the difference between the vintages. In a similar vein, if after being repeatedly being sent into the labyrinth as a sacrificial offering for the Minotaur, a single Athenian continued to escape, one could be reasonably certain the citizen knew how to escape, though one would not be able to discern the citizen’s method. These are not perfect examples, but demonstrate the general idea.

Zero knowledge proofs harness difficult mathematical problems to provide security against information leakage from the proof—the ‘zero knowledge’ property. These difficult mathematical problems can be seen in RSA, for large integer factorisation, and Diffie-Hellman, relating to the discrete logarithm problem. One-way (or *trapdoor*) functions are the embodiment of these mathematical problems, a function that is relatively easy to compute, but computationally infeasible to reverse [21], also referred to as a quality of *intractability*.

Zero knowledge protocols are classically “instances of *interactive proof systems*, wherein a prover and a verifier exchange multiple messages (challenges and responses)” [22], though there are a subset of ZKP where this interactivity is removed, reducing the number of rounds needed to establish and verify the proof. These ‘rounds’ are akin to each wine tasting, or labyrinth escape, in the previous examples. A *non-interactive* zero knowledge proof (NIZKP) allows the prover to publish a proof, in a single communication, that can be openly verified by anyone [18].

In the ‘proof’ component of a ZKP, the prover asserts a verifiable claim. One such strategy is where a *proof of knowledge* is made by the prover, examples of this are seen in the previous review of port knocking mechanics: preimage resistance of a cryptographic hash could imply proof of knowledge of the digested secret, proof of decryption could be considered proof of knowledge of the decryption key. Related is the concept of *proof of identity*, where a “person’s identity can be linked to his ability to do something and in particular to his ability to prove knowledge of some sort” [20]. Identification is a natural application for proofs of knowledge [23] and as such, a proof of knowledge protocol can be used to authenticate [24]. For example, a user could prove they know a password, without revealing any information about the password itself, to any parties privy to the conversation.

3.1.1 Identification protocols

To harness the capabilities of zero knowledge proofs for the client (or *prover*) authenticating with the port knocking server, an *identification protocol* sets out the framework for what is required of each party, how values are calculated and which values are sent as messages. Identification protocols based around zero knowledge proofs are examined in depth in [18] and [22], both works explore the protocols of Feige-Fiat-Shamir, Guillou-Quisquater and Schnorr:

– *Feige-Fiat-Shamir* The protocol uses the difficulty of extracting square roots modulo a large composite integer [22]. FFS requires a variable number of rounds (depending on desired security level), each requiring 3 messages [18].
– *Guillou-Quisquater* GQ uses the difficulty of extracting roots of a higher order [22]. GQ can be run in multiple rounds, or a single, and uses 3 messages per round [18]. GQ, at the cost of greater computation than FSS, minimises the number of interactions required, and is comprised of short, simple mathematical mechanics [18].
– *Schnorr* The Schnorr Identification Scheme harnesses the difficulty of computing discrete logs in a prime field [22]. Like GQ, Schnorr’s solution uses simple mechanics (shown in Protocol 1, please see Appendix). The protocol uses 3 messages and one round [22]. Schnorr’s has relatively low computational complexity and supports precomputation of some values [25].

Table 1 Table comparing chaotic and cryptographic properties, modified from [27]

Chaotic property	Cryptographic property	Description
Ergodicity	Confusion	The output has the same distribution for any input
Sensitivity to initial conditions/ control parameter	Diffusion from a small change in plaintext/key	A small input deviation can largely change the output
Mixing property	Diffusion from a single plaintext block	A small local deviation can largely change the whole space
Deterministic dynamics	Deterministic pseudo-randomness	A deterministic process can cause a random-like behaviour
Structure complexity	Algorithmic complexity	A simple algorithm has a very high complexity

The design goal of using only a single packet for authentication in the port knocking solution is at odds with the interactivity of the three identification protocols; specifically each requires (at least) three messages. Mao [24] also notes "a large number of interactions means a poor performance in both communication and in computation". Thankfully, a solution to this problem is outlined in RFC 8235: "Schnorr Non-interactive Zero Knowledge Proof" [26]. This RFC will form the reference basis for the remainder of this section.

3.1.2 RFC 8235

The Fiat-Shamir Transformation is a method for transforming a three-message zero knowledge proof of identity into a single message equivalent protocol. This is achieved by introducing a cryptographic hash of particular variables, in lieu of the random challenge that is issued by the verifier (message 2). As outlined in the RFC, this can be used to convert Schnorr's scheme into a non-interactive zero knowledge proof, as seen in Protocol 2.

3.1.3 Discussion

Protocol 2 (please see Appendix) will be harnessed as a method for authentication, used during prototyping in the following subsection. Considering its actions as a black-box, where randomness, a private key and shared parameters are input, the result is a proof which will be sent over the wire to be received and validated by the port knocking server. The protocol requires a secure cryptographic hash and this will form the discussion in the following section.

3.2 Chaos

Alvarez and Li [27] draws exact links between properties of chaotic and cryptographic systems. As seen in Table 1, Shannon's descriptions of simple operations, of sensitivity to initial variable changes, and of a complex output, for strong cryptographic primitives, largely correspond with behaviours in chaotic systems. The process for mixing pastry dough used as a metaphor by Shannon and Hopf actually forms the basis of a well studied chaotic system, known as the Baker's map [28].

To help illustrate the key terms in Table 1, the following definitions are included:

– *diffusion*: the spreading out of the influence of the function input over many bits of the function output [29]. For example, a single plaintext bit being distributed over the ciphertext output.
– *confusion*: the use of transformations to obscure the statistical dependencies between function input and output [29]

Lastly, it is important to clarify between some of the key terms introduced, particularly to delineate between the concepts of chaos and randomness. Truly random behaviour is non-deterministic, even if a random system is fully understood, predicting its outcome at a given future state is impossible [30]. Chaotic behaviour, in comparison, is deterministic, and every future state in the system is determined by its prior initialisation [30]. In applied cryptography, both *pseudo*-random and chaotic systems are deterministic, and used for their qualities of being *computationally unpredictable*, meaning that guessing the previous state of the system (e.g., the inputs to a cryptographic primitive) is computationally infeasible [29,30]. This is similar to how hard mathematical problems were used to hide secret information in zero knowledge proofs.

3.2.1 Chaotic systems

Chaotic systems are often result from mathematical functions, or *maps*, such as the Ikeda map and the Chirikov standard map. Mathematically, Chirikov standard map is written as:

$$\begin{cases} p_{n+1} = p_n + K \sin \theta_n \mod 2\pi \\ \theta_{n+1} = \theta_n + p_{n+1} \mod 2\pi \end{cases} \tag{1}$$

where K is control parameter, p_n and θ_n are real values between $(0, 2\pi)$. Each increment of n reflects an *iteration* of the map, much like how a hash function or ZKP uses a number of rounds [29]. The map represents a dynamical mechanical system, where the dimensions θ and p are used practically to represent position and momentum, though the important note here is that variables take on new values, on each iteration of the map, resulting in a new θp-coordinate. The constant coefficient K influences the degree of chaos exhibited by the map.

3.2.2 Chaos-based cryptographic hashes

Chaotic cryptography is an active research area, producing real-world applications including cryptographic primitives such as pseudo-random number generators (PRNG), encryption systems (both symmetric and asymmetric), and hash functions [31]. Chaotic maps and hash functions have similar characteristics [32], as explored previously in Table 1, and a number of cryptographic hash functions have been created harnessing chaotic systems.

For a formal definition, a hash function takes a *message* input, of arbitrary length, and calculates a fixed length output, known as the hash value, or a *digest*. This value is used to identify the input, acting as a fingerprint. There are a number of required properties for a hash function to be considered *cryptographically secure*, including:

- *Preimage Resistance*—given a random hash value, an attacker should never be able to find the preimage associated with the value [33]. This is why hash functions are considered as *one-way* functions—a message cannot be derived from a digest [34].
- *Second-Preimage Resistance* - given a message M_1, with hash value $H(M_1)$, an attacker should never be able to find M_2 such that $H(M_1) = H(M_2)$ [33].
- *Collision Resistance* - it should be computationally infeasible to find two (or more) different messages that hash to the same value [33].

Returning to the components being researched for prototyping, a secure cryptographic hash function is required for use in Schnorr's NIZKP. From the design goals, this ideally will be lightweight, simple, and should not require the use of third-party libraries.

3.2.3 Absolute-value chaotic hash function

The chaotic hash function proposed by [35] forms the reference basis for the remainder of this section. This implementation of a chaos-based hash function was chosen for a number of reasons:

- The chaotic map and design of the function is simple, using mostly basic mathematics, and the authors have provided pseudo-code of the algorithms involved.
- The authors have evaluated the hash using both chaotic metrics, and metrics required of secure cryptographic hashes. This includes:
  - Assurance that sensitivity to initial conditions is reflected in the hashing process.
  - Statistical analysis of confusion, diffusion.
  - Comparative testing of the statistical analysis against other hash functions, including other proposed chaotic hash functions, and against popular hash functions MD5 and SHA-1.

The hash function is *keyed*, meaning it falls into a category of cryptographic hashes that require a key *and* a message to produce the digest output. Like regular hashes, keyed variants can ensure the integrity of a message, though this is extended further to provide authentication (by use of a key) for the message. In Ref. [36], authors have utilised chaotic map, i.e., Chebyshev for broadcast authentication and chaos-based hashing. All broadcast messages were authenticated via chaos maps. A keyed hash has two main requirements to be cryptographically secure [34]: an attacker must not be able to *forge* a valid digest from a message without the key, and secondly an attacker must not be able to recover the key from message digests [34]. The hash function

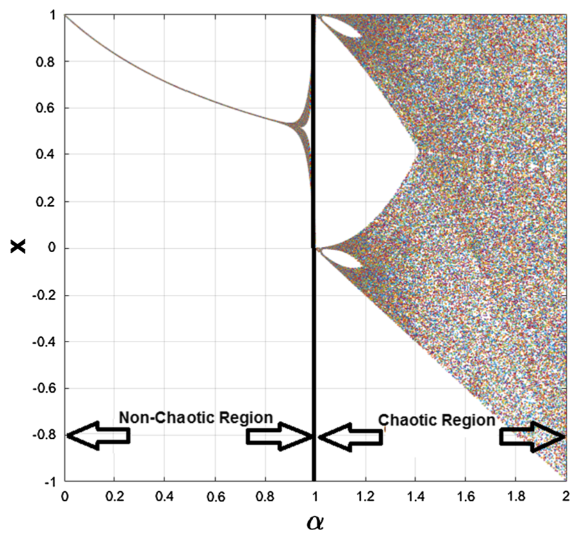


Fig. 1 Plot of Bifurcation diagram showing the chaotic region for α parameter

harnesses use of an absolute-value chaotic map given by:

$$x_{n+1} = 1 - ABS(\alpha x_n) \tag{2}$$

Equation 2 produces a series of values that chaotically vary when α is chosen to reside in the interval $1 < \alpha < 2$. The bifurcation diagram shown in Fig. 1 highlights the range of α which can be used as a key parameter for chaotic region. From Fig. 1, it is clear why one should use the interval $1 < \alpha < 2$ when random data is required. The authors' provided pseudo-code outlines how this chaotic map is converted into a hash function, which is summarised in Protocol 3 (please see Appendix).

3.2.4 Discussion

To convert Schnorr's Identification Protocol (Protocol 1, please see Appendix) into a non-interactive zero knowledge proof, a hash function is required, to act as a random oracle for both client and server. Schnorr's NIZKP, in Protocol 2, now has its required hash function. Using only mathematical operations, this hash function should require few library imports, and therefore should be fairly platform agnostic. It's keyed mode of operation will be further used to authenticate messages from client to server.

3.3 Random beacons

As discussed in Sect. 2.4, unique knock values are required to prevent an attacker re-sending a previous knock, to *replay* a previously authorised action. As port knocking solutions typically use cryptographic primitives such as hash and encryption functions, a random number is often added to the functions' inputs, ensuring that the output knock value is different for each session. This random value can be derived from methods such as iterative hashing, synchronised time clocks, or using counters. These methods require that both client *and* server have a way of reaching the same random value, i.e., it is *shared* between the two. Schnorr's NIZKP, as explored in Sect. 3.1.2 does not suffer this problem: the random value is client generated, and included in each knock, and checked by the server on arrival. Setting Schnorr's NIZKP aside for now, the design goals explicitly prevent using client state, so an alternative replay prevention mechanism is desired.

Rabin [37] introduced the idea of a random beacon, an online security service emitting a random integer at regular intervals, for public consumption, with applications in cryptographic protocols requiring trusted, shared random number access. To introduce notation, the beacon *broadcasts* a *nonce* value (a number used once), to any satellite clients, who optionally may further process the broadcast through *extractor functions*. Formally, a list of requirements for a beacon service is provided by [38]:

- *unpredictable*: an adversary should not be able to predict any information about the nonce prior to its broadcast
- *unbiased*: the nonce should be statistically close to uniform, random information
- *universally sampleable*: any satellite client should be able to harvest (or extract) the nonce
- *universally verifiable*: the beacon can be verified to be unknown to any party prior to its broadcast

3.3.1 Sources of randomness

A number of solutions are present in the literature for random beacons. Jiwa et al. [39] suggests that Network Time Protocol (NTP) can be used as a beacon. As an example, the hash of an NTP received timestamp (within a safe margin of precision) could produce a pseudo-random nonce. This is similar to the port knocking implementation explored in Sect. 2.4, where clock synchronisation can be considered a shared state between client and server. Again, to avoid synchronisation issues between port knocking client and server, per the design goals, this is not an applicable solution for the prototype. Furthermore the time-measurements issued by an NTP service are likely unverifiable.

Clark and Hengartner [40] propose using the price of publicly listed financial instruments, taken at closing time, as a source of random data. The authors note this approach has previously been used in cases including committee nomination, proof of work cryptographic puzzles, and in two public elections within North America. The proposed solution collects closing prices from a number of stocks and uses a extractor function (with elements of a PRNG) to harvest quantifiable entropy results from the financial beacons. Lee et al. [41] also provide an implementation using stock indices as random beacons. Their paper details considerations on particular stocks to harvest, which exchanges to use and associated timezone factors. Both papers note that there are historic examples of price manipulation in financial markets, a method by which the security of the random beacon could be adversely affected by a malicious actor. Such a scenario would constitute a breach of the unpredictability property of the random beacon. Lee et al. [41] suggest a greater problem may be posed by trusting the price reporting website (e.g., Bloomberg) to accurately reflect market values. Bonneau et al. [38] highlights that limited market opening hours will result in lack of beacon availability.

Lenstra and Wesolowski [42] discusses the newly introduced NIST random beacon, a public service providing a 512-bit nonce, every minute, of true-random data, generated using “quantum mechanical phenomena”. This sounds ideally suited for the purpose, however, the authors note that while the service is extensively documented, there is no assurance to consumers that the numbers are generated as advertised. Simply put, again, the beacon is unverifiable. Bonneau et al. [38] treats NIST’s reputation as a trusted party in this context with a heavy dose of scepticism, alluding to the discovery of a backdoor in the NIST-published Dual_EC_DRBG cryptographic standard.

Relating back to the beacon requirements, these solutions are all predominantly unverifiable, and as a result, can’t be provably *unpredictable*, a desired quality. To address this problem, a number of different beacons can be used, where their results are combined or evaluated in a fashion that reduces the likelihood of tampering. Some of the methods used to tackle dishonest beacons include:

XOR nonces [43]: in this solution each beacon service is polled, and the resulting nonces are XOR’ed together. If the beacon sources are incapable of influencing each other (perhaps they can’t witness each other’s broadcasts), then this method does reduce the effectiveness of tampering, by making the outcome more unpredictable. However, if the beacons *are* capable of influencing each other, the authors note that the last beacon to be polled could calculate the penultimate XOR result and alter its broadcast accordingly to influence the final outcome. In such a scenario, the solution is ineffective and redundant.

Round-robin nonces [43]: this solution (also discussed in [44]) simply rotates which sources are used by using each beacon for a fixed time-period, before moving onto the next. Unpredicability of the outcome is proportional to the ratio of dishonest beacons in the pool, meaning if at least one beacon is honest, then the round-robin approach does indeed makes the overall outcome more unpredictable. Unfortunately, this would require a shared state between the client and server, namely the position of the round, disagreeing with the design goals.

Hashing nonces [43]: the collection of nonces from different beacons could be hashed in some manner, the authors suggest that a hash of the concatenation of each nonce could be taken. The authors then argue, an adversary with a large amount of computing power at their disposal could use bruteforce techniques to find preimages of the hash function with certain qualities (e.g., $H(m)$ starts with ’00’) resulting in the compromise of the other nonces in the pool. For these reasons, hashing nonces is considered by the authors as equal or less secure than the XOR strategy.

Delayed evaluation [45]: as discussed previously, a dishonest beacon can alter its nonce in an attempt to influence the other nonces it is combined with, to try and influence the final outcome. Variable delay functions (VDF) are processes designed to strategically *slow down* calculation of some task, and hence can be used to slow down the process of evaluating the pool of nonces. If for example, two independent beacons publish nonces in an hourly window, the delay function could be set to prolong the time it takes to combine these nonces. If the delay was set to a long enough period, neither beacon would have the time to derive a malicious nonce, before the hourly window had transpired and new nonce values were expected to be broadcast. Lenstra and Wesolowski [42] contains further discussion on VDFs and their applications.

These anti-tampering methods provide options for combining multiple random beacons, where the issue with trusting an single-source provider of random data appears to largely concern how predictability can be avoided, given that solutions are largely unverifiable, and could be tampered with. The proceeding section covers the random beacon chosen for the prototype, that is transparent, distributed, an accordingly conveys a higher degree of trust. It is worth noting however, that preferably if the literature had provided a suitable alternative, applicable to the design goals for the prototype, the combination of *multiple* random beacons would be the preferred option, as discussed in the evaluation.

3.3.2 Bitcoin random beacon

Blockchains are distributed, immutable ledgers of transactions used to record and authenticate activities of involved parties. Bitcoin is one such instance of blockchain technology known as a *cryptocurrency*, offering openly decentralised, self-governed currency transactions, backed by secure cryptographic protocols. Bitcoin transactions are collected into blocks, where within each block a Merkle Root is calculated and recorded in the block header. The Merkle Root involves hashing each transaction in the block in a manner that ensures that the entire contents of the block, including its ordering, is recorded to protect the integrity of the block's contents. Once a block is filled with transactions, the hash of the *previous block* on the chain is added into its header, forming the 'chain' of collections of transactions, where each link ensures the integrity of its predecessor [46].

The job of calculating the latest block on the chain is tasked to miners, who are rewarded with Bitcoin currency for their efforts. Alongside the aforementioned hashing calculations, miners must also solve a proof of work puzzle for each new block—the block's hash digest must begin with a number of zeroes—a task requiring millions of hash calculations. If multiple chains are broadcast to the Bitcoin network the longest is chosen, meaning it has been created with the most work, and therefore is the *consensus* reached by the mining community. This ensures that in order to tamper with the blockchain, an attacker would require computational power surpassing the rest of the mining community [46].

The chosen random beacon for the prototype is taken from [38], which forms the reference source for the remainder of this section. The paper proposes that the random values resulting from proof of work computations in the Bitcoin network make each new block mined on the network a suitable random beacon (please see Protocol 4, Appendix).

4 Experimentation

4.1 Prototype I: ZKP and chaos

The first prototype introduced here combines the work of Schnorr's Non-Interactive Zero Knowledge Proof (NIZKP) and the Chaos-based Keyed Hash Function, from 3.1 to 3.2 in the preceding subsection. Prototype I will display a large amount of the general structure ahead of its successors.

4.1.1 Setup

The setup of a client and server is the only out-of-band process, used to derive the secrets and configurations required for port knocking operations. The subroutine achieving this produces a *profile* JSON file each for client and server, housing cryptograhic parameters and configuration settings. The profile contents include:

- Schnorr NIZKP parameters (refer to Sect. 3.1.2)
    - Group parameters p, q, g are generated using a call to the `openssl` library. Specifically this uses the `dsaparam` module, to generate DSA parameters, which are applicable for the NIZKP as outlined in [26]. DSA key-length is set to 2048-bit.
    - Private key a is randomly selected from the range $[0, q-1]$ and is used to derive public key $A \equiv g^a$. Note: the only difference between the client and server profiles generated is the

  - absence of a in the server profile, as the ZKP proves knowledge of a.
  - 8-bit command ID - for future development to support multiple commands under a single profile. In the original protocol, these were refered to as *UserID*, and *OtherInfo*.
- Private hash key: a float number in the range $(-1, 1)$, with 256-bit precision, for an associated 256-bit keyspace. Implemented using the `mpmath` 3rd-party Python library.
- Server port number: randomly generated during setup, and fixed throughout all exchanges.
- Command: user specified shell command to run upon successful client authentication.

On a modern laptop, generation of the profiles was near instantaneous, and each was sized at 3KB. The profiles should be securely transferred and housed on the client and server machines.

4.1.2 Chaos-based hash function

The hash digest length was increased to 256-bit, as was required by RFC 8235 [26] for compatability with the NIZKP: "The bit length of the hash output should be at least equal to that of the order q of the considered subgroup." This level of precision necessitated that the `mpmath` library be used, above the standard float data types that ship with Python. The horsepower required to manage calculations with these floats, with up to $\log(2^{256})/\log(10) \approx 78$ decimal places, had a dramatic affect on the hashing speeds, with the hash taking over 20 s to calculate the required digest for the NIZKP protocol. Issues with chaotic map computations of floating-point numbers were noted by [32], though were not expected to be this severe. A large factor in the sluggishness of the hash function was the number of iterations to perform, or in terms of the chaotic map, the time parameter t. From the authors' paper, little guidance was set for how to select this parameter, $t = 10,000$ was one such baseline used for testing, however this proved unachievable in practical time. A further factor in this speed is undoubtedly increasing the hash's bit length. Lastly, as the hash was implemented in Python, a high-level language, time savings could further be gained by implementing the final version of the port knocking application in a language closer to the client hardware.

4.1.3 Client actions

In Schnorr's NIZKP, a hash is taken of the public key A, the random walk $V \equiv g^v$, where v is randomly generated by the client, and other, non-essential parameters. The resulting digest c, accompanied by $r \equiv v - ac$, form the proof of knowledge, or knock in this case, as the pair of concatenated values $c||r$, that are then transmitted to the port knocking server in the payload of a UDP datagram.

UDP was chosen as the transport protocol for sending the values, as TCP connection-based overhead and interactivity were not deemed necessary nor applicable, respectively, to minimalist design goals. ICMP traffic could be an alternative, though nothing was found in the literature to support this approach, whereas [12,47,48] provide justifications for choosing UDP. Python's native `socket` library was used to send the knock packet, where the server IP and destination port values are both taken from the client profile, the former being user input from `stdin`, and the latter being randomly generated.

4.1.4 Server actions

The server parses traffic using `scapy`, a 3rd-party Python library that provides an interface to lower-level `libpcap` functionality. The open-source `libpcap` library for network traffic capture operates on Linux systems, allowing packets to be inspected before firewall rules [7], and is "amongst the most widely used [APIs] for network packet capture". Scapy is used to continually sniff the server's network interface. Collected traffic is then subjected to conditional rulesets designed to filter out traffic not meeting ZKP criteria:

1. Traffic must be destined for the server port, as setup in the client profile.
2. Traffic must be UDP, with an integer payload.
3. The payload is of length equal to the hash digest length, plus an integer $r \mod q$, padded with zeroes to fix this length.

If a packet is sniffed meeting these criteria, then the Schnorr NIZKP Protocol (see 3.1.2) outlines the mathematical checks performed to validate the proof, computationally, this involves:

1. Calculate c, v, by splitting the packet payload into their appropriate lengths.

2. Calculate $V \equiv g^r A^c$ using values from Step 1, and the client profile.
3. Check whether the received c is equal to: $\mathrm{H}\,(V \| A \| \mathrm{CommandID})$

If the proof is successful, the client is authenticated, and their requested shell command is executed. Though not implemented, the protocol allows for accompanying the proof with further data, this could be used to facilitate multiple command options for a single client, and multiple users.

4.1.5 Discussion

Prototype I uses third-party libraries only for floating-point computations, and packet sniffing from the wire. Theoretically, a powerful pocket calculator could generate the proof required to authenticate, and it could be sent via any number of online services for testing network connectivity. It could be argued that reducing dependencies such as 3rd-party libraries use will reduce threat vectors from exploitation of those libraries. The prototype is further well suited for application on devices with limited ability to maintain updated libraries. Replay protection is a corollary of mixing randomly chosen v, and resulting V, in some form, into c and r, ensuring each time a knock is sent, each element in the proof is derived from a nonce.

Returning to the design goals, the aim of having the server precompute acceptable knock values is entirely missed. Instead, this prototype requires the server to check a potential client proof, requiring two exponentiation operations in the group setting [26], and a hash function execution. This opens a serious attack vector: by sending data to a suspected port knocking service port, an attacker can force the server to perform computations, potentially blocking out valid client knocks. As per [49], "processing required to verify the [knock] should be minimal and not introduce a [DoS] vector.". Filtering options are limited for preventing such attacks, as knock values are pseudo-random, possessing no identifiable characteristics other than length, contradicting goals from [49]: "unauthorized packets should be rejected as early as possible to reduce attack surface and decrease server side processing."

To combat this, a traffic rate limiter could be enforced (the `scapy` sniffer offers such a feature) though this could in turn enable an attacker to purposely trigger the rate limiter to block out legitimate clients. Alternatively, the server's sniffer could establish a sort of reputational filter, ignoring the traffic from IPs that have sent invalid knocks, though this could be circumvented if the attacker spoofed their IP. Hopefully these scenarios illustrate why both the *only precomputation* and *only key-based authentication* were included as design goals. Suffice to say, this prototype does not fare well against denial of service attacks. DoS mitigation could be supplied by network and host monitoring solutions, such as IDPS, though this should not be required of a port knocking implementation.

4.2 Prototype II: chaos and random beacons

The second prototype explored in this subsection forgoes zero knowledge proofs, and instead adopts a random beacon service, as described in 3.3. The chaos-based hash functionality is further retained, though this time it is used to parse the random beacon. The client and server profiles are setup as previously, though without the parameters for Schnorr's NIZKP. Largely, the code is replicated from the previous prototype, with modifications outlined in the following.

4.2.1 Blockchain-based random beacon

The Bitcoin random beacon protocol, as shown in Sect. 3.3.2, pulls information from a website providing updates on Bitcoin's blockchain. The Python code for implementing this feature is taken from [50], with small changes, including a change of website to the blockchain API provided by [51], which for this purpose requires only a single HTTPS GET request using the Python native `request` library, to a URL of the website's blockchain API. This API replies, offering information on the latest published block in JSON format. From there, the block header, and block header hash, are extracted and combined through a pairwise `OR`. The resulting string is then passed through the chaos-based keyed hash function, resulting in a 256-bit beacon value. Of note, if the hash function used were *not* keyed, then the attacker would be able to recreate the beacon output. Given that the key used in the hash function is only available to client and server, as per Protocol I's setup, this means the beacon values are shared, random and private. The *keyed* hash function could be used to combine multiple random beacons, for greater security (see Sect. 3.3.1, though this is beyond the scope of the demonstration here.

After a new beacon has just been received, the beacon service sleeps for a fixed number of minutes, which can be changed to preference in accordance with block production speeds varying roughly around the 1 per 10 minutes mark [38]. Following this wait, the beacon will check more frequently, until the data pulled from the API differs, signalling a recalculation of the knock to expect from a client.

4.2.2 Client operations

For this prototype, the UDP knock payload is defined as H_k(beacon||command), where k and H are the key and chaos-based hash implemented in the previous prototype. As discussed in the port knocking mechanisms literature review, hashes are one-way functions that can provide a proof of identity: the attacker can only generate this payload if they are privy to the secret key, which is securely shared in out of band setup. Replay protection in this instance is provided by the freshness derived from the beacon. After processing, the beacon's values are pseudo-random, so the only instance in which the hash function receives two identical messages would be where the blockchain API reports an identical block and block hash. Therefore, the only instance in which a knock-collision occurs would result from either this scenario, or a weakness in the hash function. The security level resulting from a 256-bit keyspace for the chaos-based hash function ensures this is unlikely. Once constructed, the UDP payload is sent out as previously.

4.2.3 Server operations

The port knocking server's operations include periodically harvesting random beacon data, converting this data into the knock it expects to see from the client, and monitoring the network to detect whether this knock has been received. These responsibilities require a degree of concurrency, for example the server must retain its listening abilities whilst at the same time calculating what the proceeding knock should look like, which uses the overwhelmingly slow chaos-based hash function. To handle these tasks in parallel, the native Python `multiprocessing` library is used to setup individual processes for traffic monitoring, beacon harvesting, and authorised command handling. Special variables handled by a `multiprocessing Manager` are shared between the processes, for instance to signal that the client has successfully authenticated.

4.2.4 Discussion

Once a beacon has been harvested and processed, the server is left with a string value to listen for on the network, which if found, signals authentication, and subsequently the user's command is authorised. To enable multiple commands, a new H_k(beacon||command) is calculated for each, per beacon, and accordingly listened for. In comparison with the first protocol, and in the context of the design goals, this variant is vastly more simple, and has managed to eschew the denial of service attack vectors the latter was vulnerable to. Where previously the server performed exponential calculations and hashing operations to verify the authenticating data, this prototype need only check whether strings are equal. By removing the Schnorr framework, only a single private key is required (for the keyed hash), and the `openssl` library can be forgone. The introduction of a trusted third party (the blockchain API), is an obvious security detractor. Further examination of this aspect will follow later, in the evaluation section.

4.3 Prototype III: crucible

The final prototype introduced by this paper differs largely from the previous iterations. As a consequence of research leading up to this section, Prototype III, henceforth named Crucible (a namesake derived from its blending of ideas, much like elements in a furnace) draws its motivations from zero knowledge proofs, and chaos-based cryptography, but instead replaces both previously implemented components with dedicated off-the-shelf, cryptographic alternatives, that are already established in practice.

For authentication purposes, the previous review of zero knowledge proofs aimed to lead development towards a practical proof of identity scheme which could be used to prove knowledge of a secret key. Crucible pursues this line of thinking, but instead uses a password-based key derivation function (PBKDF) to prove knowledge of a *password*. In doing so, the server does not possess the password itself, only its hash. In both previous prototypes, the run-time of the chaos-based keyed hash severely dampened results, instead, in Crucible, it will be replaced with a modern keyed hash alternative.

4.3.1 Password-based key derivation

A key derivation function converts a secret value (such as a master key, passphrase, or password) into a secure cryptographic key [34]. Password-based key derivation functions (PBKDF) are the family of such solutions aimed at converting potentially weak user supplied passwords into a keys that are specifically designed to be resistant to cracking attempts, making them well suited for storing user credentials on a server, whilst limiting the exposure to users if that server were to be compromised [52].

As from the discussion on what comprises a secure cryptographic hash in Sect. 3.2.2, a PBKDF requires all the qualities of preimage, second-preimage, and collision resistance. In addition to this, a PBKDF needs resistance to lookup table attacks (e.g., rainbow tables), CPU-optimised cracking, hardware-optimised cracking (e.g., GPUs, FPGAs and ASICs), amongst other criteria, as established by the 2015 Password Hashing Competition [52]. Argon2 was the winner of this competition, and is selected for purpose in Crucible. Further details on Argon2 can be found in the IETF draft [53] and design paper [54].

The prototype uses Python's native `passlib` library, updated with installation of the `Argon2` package. This specifically uses the Argon2i variation, recommended for password hashing purposes, as opposed to other KDF operations [53]. The hash parameters chosen for this prototype set Argon2 to use 20 rounds, 256 MB ram, and 2 threads of parallelism, producing a hash in under 2 s on a modern laptop. Salt is set to a fixed value (explained later in Sect. 5.1.3), and with the chosen parameters is stored alongside the digest, in a single hash string. These settings should be chosen in implementation to maximise computation efforts under the used hardware environment.

The setup process is similar to the preceding prototypes, where instead Argon2 replaces key generation procedures: the user is asked to input their chosen password (which they are to memorise), the hash of which is then stored only on the port knocking server. In addition to memorising the password, with Crucible the user needs to know only the IP address on which to knock, and the command name to execute. In this manner, Crucible is *stateless*, whereby a user can download the client application, and perform port knocking, without needing access to secret keys or other parameters, i.e., following installation of a profile on the server, a client profile (as per previous prototypes) is not required.

4.3.2 Keyed hash

With the client able to derive a key from their chosen password, Crucible follows Prototype II in using a keyed hash to digest a random beacon. In preference of the chaos-based keyed hash already explored previously, an established hash function is taken from the literature.

BLAKE2 was chosen to perform the task of hashing operations in Crucible, being amongst the fastest secure hash functions available, and the most-popular non-NIST standard hash [34]. Unlike alternatives Siphash, and SHA3, BLAKE2 is resistant to side-channel attacks, where an attacker has access to RAM and registers [34]. Further, unlike SHA3, BLAKE2 has native support for keyed mode hashing, without additional construction. More information on BLAKE2 can be found in [55].

The particular variant BLAKE2b, was chosen for 64-bit optimisation in the test environment. The `pyblake2` 3rd-party Python library [56] was used for the Python 2.7 implementation (referenced on the authors' website [57]), however the native `hashlib` supports BLAKE2 for later versions.

4.3.3 Setup

Ahead of port knocking operations, the client must generate a profile to leave with the server, used by the server to know which knocks to listen for. The generation process firstly prompts the user for a password, and the user is then directed to submit a number of commands for the server to execute on successful authentication. Each command must be named, and along with the password and IP of the server, each command name must be memorised. Following this, per the previous prototypes, the client profile is serialised in JSON and saved in a text file, to remain with the server.

4.3.4 Client operations

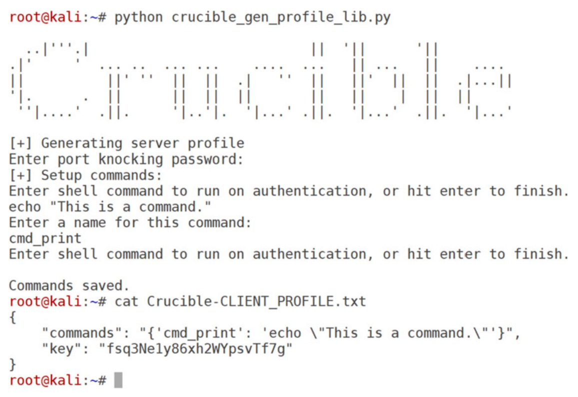
To execute a command on the port knocking server, a remote user runs the client Python script, and the following operations are performed:

1. The client supplied password is run through Argon2 to generate the associated key:
   key = Argon2b(password).
2. The client script pulls down and calculates the most recent random beacon from the blockchain API.
3. The beacon and client supplied command are concatenated, and hashed with BLAKE2, using a keyed mode of operation:
   knock = $\text{BLAKE2b}_{\text{key}}(\text{beacon}||\text{command})$
4. The server's port number listening to the knocks is derived similarly to the key, though instead the command is replaced with "0". The first two bytes of the resulting hash are then converted into an integer port number, this combined with the IP provide the destination to which the knock is sent.

The key, beacon, and knock at i are calculated as:

$$\begin{aligned} \text{key} &= \text{Argon2i}\,(\text{password})\,, \\ \text{beacon} &= \text{BLAKE2b}_{\text{key}}\,(\text{blockHeader} \\ &\quad + \text{blockHeaderHash})\,, \text{knock}_i \\ &= \text{BLAKE2b}_{\text{key}}\,(\text{beacon}||\text{command}_i) \end{aligned}$$

where i represents the chosen command for the particular knock session (the server will listen for all possible values of i).



```
root@kali:~# python crucible_gen_profile_lib.py

  ..|'''.|                          ||  '||      '||
.|'     '  ... ..  ... ...   ....  ...   || ...   ||    ....
||          ||' ''  ||  ||  .|   '' ||   ||'  ||  ||  .|...||
'|.      .  ||      ||  ||  ||      ||   ||    |  ||  ||
 ''|....'  .||.     '|..'|.  '|...' .||.  '|...'  .||.  '|...'

[+] Generating server profile
Enter port knocking password:
[+] Setup commands:
Enter shell command to run on authentication, or hit enter to finish.
echo "This is a command."
Enter a name for this command:
cmd_print
Enter shell command to run on authentication, or hit enter to finish.

Commands saved.
root@kali:~# cat Crucible-CLIENT_PROFILE.txt
{
    "commands": "{'cmd_print': 'echo \"This is a command.\"'}",
    "key": "fsq3Ne1y86xh2WYpsvTf7g"
}
root@kali:~# 
```

Fig. 2 Crucible setup: generating a client profile

4.3.5 Server operations

The structure for the server operations is largely similar to Prototype II: there are a number of interrelated tasks the server needs to perform concurrently, managed through Python's `multiprocessing` library:

– The server periodically checks the blockchain API for new values, and stores harvested beacons in a `multiprocessing Manager` dictionary, to enable access by other processes. Once a new beacon is harvested, a boolean in the dictionary is flipped, signalling that the traffic sniffing process needs to update the valid knocks it listens to.
– Whenever a new beacon is found, for each command in the client profile a new valid knock string must be calculated (per Client Operation 3). Once generated, each knock string is sent to the listening process, using `scapy`. This listening process acts on the port number generated per client Operation.
– The listening process, checks each received UDP packet, of appropriate length, on the correct port, against the calculated valid knock values. If a match is found, the relevant command is sought and executed.

4.3.6 Discussion

The random port number derived from the beacon serves two purposes: firstly, it may make the job of an attacker listening on the wire slightly more difficult, as there is one less defining traffic characteristic to filter by; secondly, it improves usability, as the user no longer needs to memorise the port number to knock against. This change from Protocol II is made possible by the enormous difference in hash run-times.

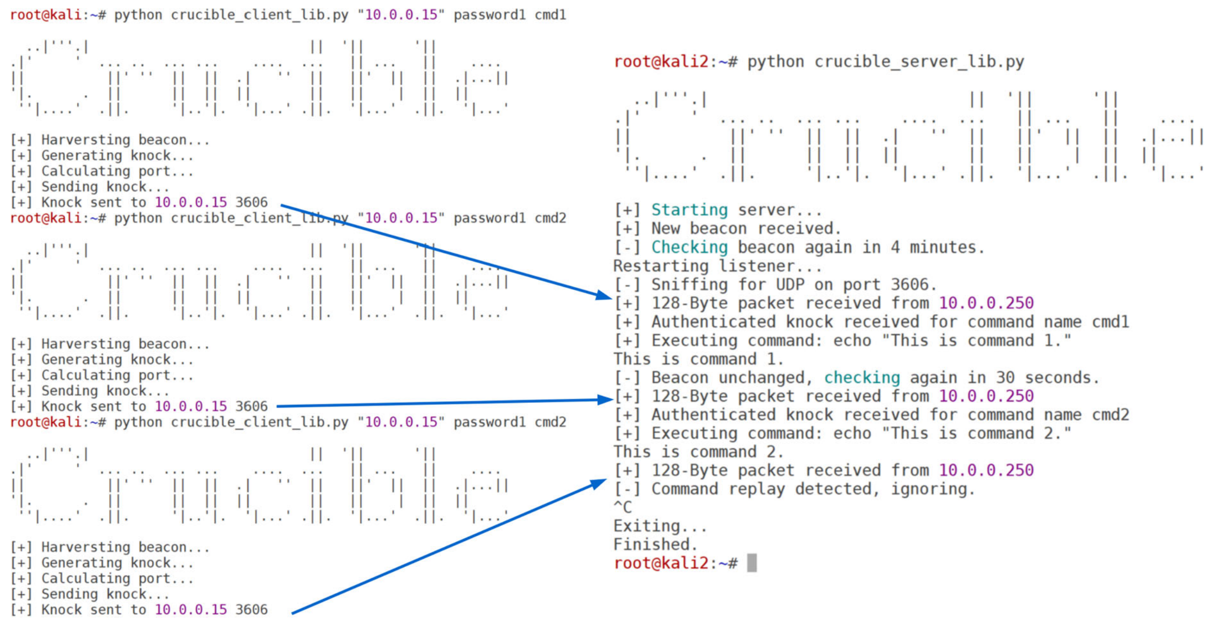
5 Evaluation

To begin using Crucible, the client must generate a profile and share this with the server. Figure 2 shows this process, with the generated profile printed out. The key is derived from the user's password using Argon2, as per Sect. 4.3.1.

Figure 3 shows typical usage of Crucible. The client runs the Python script, passing in the IP, password and command parameters. There is also an interactive mode alternative for this operation. The client executes three commands, “cmd1”, “cmd2” and “cmd2”. As the latter command has already been executed by the server, it is ignored for replay protection. These commands will become available again when a new random beacon is sourced.

Figure 4 is a traffic capture of traffic emanating from the client machine as a result of port knocking with Crucible. It can be seen that Crucible does indeed only require a single packet for authentication and command authorisation. Further, the beacon data collected from the blockchain API is seen to be protected via HTTPS.

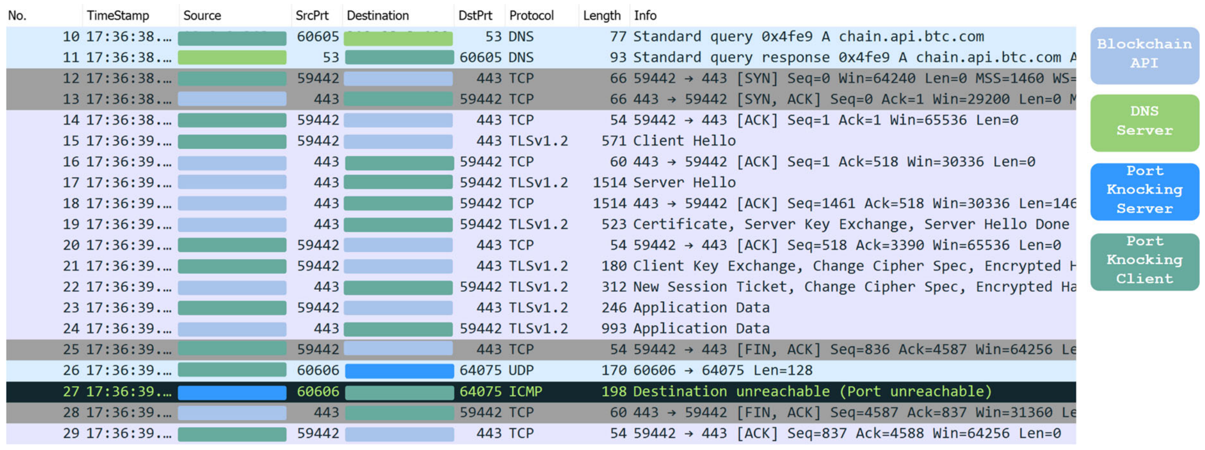
The knock itself, as seen in Fig. 5, is a single 512-bit BLAKE2 hash sent via UDP, both the payload contents and the destination UDP ports are pseudo-randomly derived, the source port is ephemeral and chosen by the client OS.



```
root@kali:~# python crucible_client_lib.py "10.0.0.15" password1 cmd1

  ..|'''.|                            ||  '||       '||
.|'     '  ... ..  ... ...    ....   ...   || ...    ||    ....
||          ||' ''  ||  ||  .|   ''   ||   ||'  ||   ||  .|...||
'|.      .  ||      ||  ||  ||        ||   ||    |   ||  ||
 ''|....'  .||.     '|..'|.  '|...'  .||.  '|...'   .||.  '|...'

[+] Harversting beacon...
[+] Generating knock...
[+] Calculating port...
[+] Sending knock...
[+] Knock sent to 10.0.0.15 3606
root@kali:~# python crucible_client_lib.py "10.0.0.15" password1 cmd2

  ..|'''.|                            ||  '||       '||
.|'     '  ... ..  ... ...    ....   ...   || ...    ||    ....
||          ||' ''  ||  ||  .|   ''   ||   ||'  ||   ||  .|...||
'|.      .  ||      ||  ||  ||        ||   ||    |   ||  ||
 ''|....'  .||.     '|..'|.  '|...'  .||.  '|...'   .||.  '|...'

[+] Harversting beacon...
[+] Generating knock...
[+] Calculating port...
[+] Sending knock...
[+] Knock sent to 10.0.0.15 3606
root@kali:~# python crucible_client_lib.py "10.0.0.15" password1 cmd2

  ..|'''.|                            ||  '||       '||
.|'     '  ... ..  ... ...    ....   ...   || ...    ||    ....
||          ||' ''  ||  ||  .|   ''   ||   ||'  ||   ||  .|...||
'|.      .  ||      ||  ||  ||        ||   ||    |   ||  ||
 ''|....'  .||.     '|..'|.  '|...'  .||.  '|...'   .||.  '|...'

[+] Harversting beacon...
[+] Generating knock...
[+] Calculating port...
[+] Sending knock...
[+] Knock sent to 10.0.0.15 3606
```

```
root@kali2:~# python crucible_server_lib.py

  ..|'''.|                            ||  '||       '||
.|'     '  ... ..  ... ...    ....   ...   || ...    ||    ....
||          ||' ''  ||  ||  .|   ''   ||   ||'  ||   ||  .|...||
'|.      .  ||      ||  ||  ||        ||   ||    |   ||  ||
 ''|....'  .||.     '|..'|.  '|...'  .||.  '|...'   .||.  '|...'

[+] Starting server...
[+] New beacon received.
[-] Checking beacon again in 4 minutes.
Restarting listener...
[-] Sniffing for UDP on port 3606.
[+] 128-Byte packet received from 10.0.0.250
[+] Authenticated knock received for command name cmd1
[+] Executing command: echo "This is command 1."
This is command 1.
[-] Beacon unchanged, checking again in 30 seconds.
[+] 128-Byte packet received from 10.0.0.250
[+] Authenticated knock received for command name cmd2
[+] Executing command: echo "This is command 2."
This is command 2.
[+] 128-Byte packet received from 10.0.0.250
[-] Command replay detected, ignoring.
^C
Exiting...
Finished.
root@kali2:~# 
```

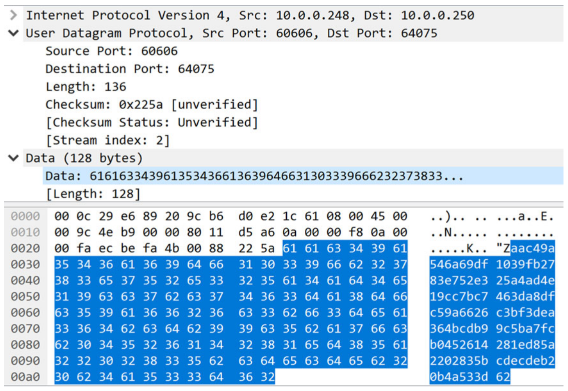
Fig. 3 Normal client and server operations of Crucible



No.	TimeStamp	Source	SrcPrt	Destination	DstPrt	Protocol	Length	Info
10	17:36:38.…	■	60605	🟩	53	DNS	77	Standard query 0x4fe9 A chain.api.btc.com
11	17:36:38.…	🟩	53	■	60605	DNS	93	Standard query response 0x4fe9 A chain.api.btc.com A
12	17:36:38.…	■	59442	□	443	TCP	66	59442 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=
13	17:36:38.…	□	443	■	59442	TCP	66	443 → 59442 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 M
14	17:36:38.…	■	59442	□	443	TCP	54	59442 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
15	17:36:39.…	■	59442	□	443	TLSv1.2	571	Client Hello
16	17:36:39.…	□	443	■	59442	TCP	60	443 → 59442 [ACK] Seq=1 Ack=518 Win=30336 Len=0
17	17:36:39.…	□	443	■	59442	TLSv1.2	1514	Server Hello
18	17:36:39.…	□	443	■	59442	TCP	1514	443 → 59442 [ACK] Seq=1461 Ack=518 Win=30336 Len=146
19	17:36:39.…	□	443	■	59442	TLSv1.2	523	Certificate, Server Key Exchange, Server Hello Done
20	17:36:39.…	■	59442	□	443	TCP	54	59442 → 443 [ACK] Seq=518 Ack=3390 Win=65536 Len=0
21	17:36:39.…	■	59442	□	443	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Encrypted H
22	17:36:39.…	□	443	■	59442	TLSv1.2	312	New Session Ticket, Change Cipher Spec, Encrypted Ha
23	17:36:39.…	■	59442	□	443	TLSv1.2	246	Application Data
24	17:36:39.…	□	443	■	59442	TLSv1.2	993	Application Data
25	17:36:39.…	■	59442	□	443	TCP	54	59442 → 443 [FIN, ACK] Seq=836 Ack=4587 Win=64256 Le
26	17:36:39.…	■	60606	🟦	64075	UDP	170	60606 → 64075 Len=128
27	17:36:39.…	🟦	60606	■	64075	ICMP	198	Destination unreachable (Port unreachable)
28	17:36:39.…	□	443	■	59442	TCP	60	443 → 59442 [FIN, ACK] Seq=4587 Ack=837 Win=31360 Le
29	17:36:39.…	■	59442	□	443	TCP	54	59442 → 443 [ACK] Seq=837 Ack=4588 Win=64256 Len=0

□	Blockchain API
🟩	DNS Server
🟦	Port Knocking Server
■	Port Knocking Client

Fig. 4 Wireshark traffic capture of the client knocking process. Packets 10–12 show the client’s DNS request for the Blockchain API’s IP. Packets 12–24, and 28–29 show the random beacon retrieval over HTTPS. Packet 26 is the client knock. Packet 27 is a closed-port ICMP reply, per RFC 792 [58]



```
> Internet Protocol Version 4, Src: 10.0.0.248, Dst: 10.0.0.250
v User Datagram Protocol, Src Port: 60606, Dst Port: 64075
     Source Port: 60606
     Destination Port: 64075
     Length: 136
     Checksum: 0x225a [unverified]
     [Checksum Status: Unverified]
     [Stream index: 2]
v Data (128 bytes)
     Data: 616163343961353436613639646631303339666232373833...
     [Length: 128]
0000  00 0c 29 e6 89 20 9c b6  d0 e2 1c 61 08 00 45 00   ..).. .. ...a..E.
0010  00 9c 4e b9 00 00 80 11  d5 a6 0a 00 00 f8 0a 00   ..N..... ........
0020  00 fa ec be fa 4b 00 88  22 5a 61 61 63 34 39 61   .....K.. "Zaac49a
0030  35 34 36 61 36 39 64 66  31 30 33 39 66 62 32 37   546a69df 1039fb27
0040  38 33 65 37 35 32 65 33  32 35 61 34 61 64 34 65   83e752e3 25a4ad4e
0050  31 39 63 63 37 62 63 37  34 36 33 64 61 38 64 66   19cc7bc7 463da8df
0060  63 35 39 61 36 36 32 36  63 33 62 66 33 64 65 61   c59a6626 c3bf3dea
0070  33 36 34 62 63 64 62 39  39 63 35 62 61 37 66 63   364bcdb9 9c5ba7fc
0080  62 30 34 35 32 36 31 34  32 38 31 65 64 38 35 61   b0452614 281ed85a
0090  32 32 30 32 38 33 35 62  63 64 65 63 64 65 62 32   2202835b cdecdeb2
00a0  30 62 34 61 35 33 33 64  36 32                     0b4a533d 62
```

Fig. 5 Wireshark traffic capture of the client knock

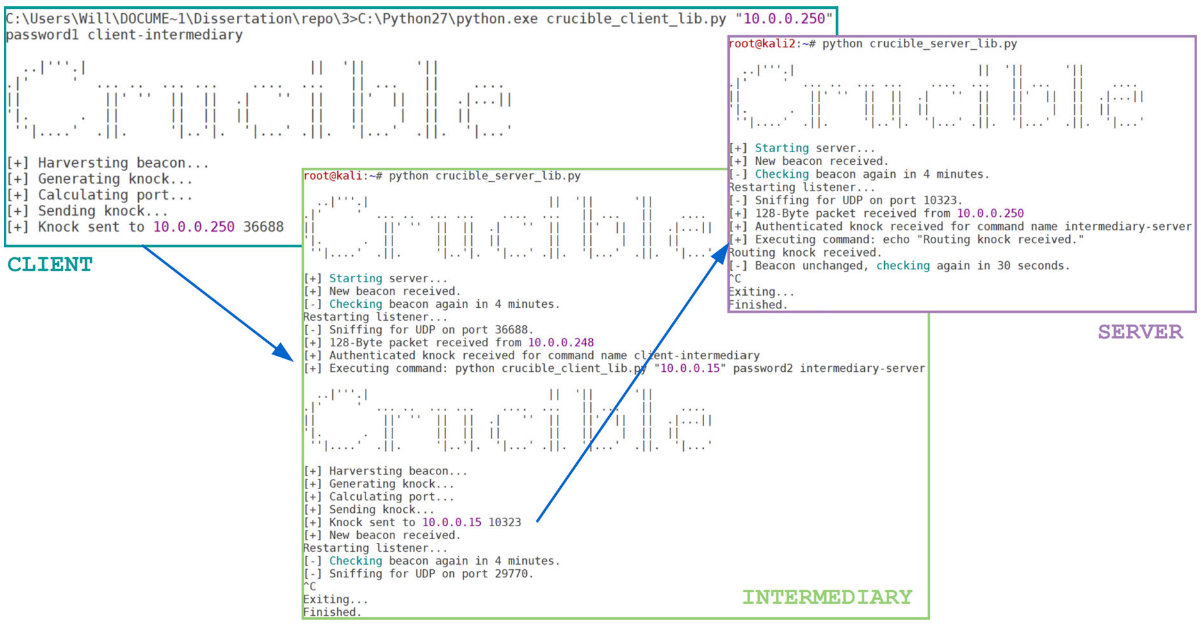
Figure 6 demonstrates how Crucible avoids the NAT problems faced by port knocking solutions. This problem arises if a port knocking client attempts to authenticate with a server whose IP address has been translated, for example, by NAT [59], proxies, or a VPN [6]. As a result, the knock packet sent by the client will not reach the intended destination. In such a scenario, Crucible should be installed on each intermediary host, and is setup with a command to spawn a new client process, and knock on the next host in the chain, until the command reaches its final destination.

5.1 Attack modelling

In this section a range of potential attacks against the operations of Crucible are considered, exploring what capabilities an adversary may have, and what the ramifications of each attack are. Aumasson [34] outlines the goals of attack modelling, which are paraphrased as follows:

- To help set requirements for future protocol design.
- To provide users guidelines on whether a protocol will be safe to use in their environment.
- To provide clues for analysts keen to find weaknesses in the protocol, as part of the security process, so they can determine whether a given attack is valid.

Unless otherwise specified, the attack assumptions this section operates under are that the attacker is positioned per Fig. 7. This simple diagram may seem pointless to include, but it helps formalise firstly that all networks Crucible operates across are considered untrusted. Secondly, the diagram highlights the potential extent of attacker capabilities under a worst-case scenario – full traffic access, and interdiction.



```
C:\Users\Will\DOCUME~1\Dissertation\repo\3>C:\Python27\python.exe crucible_client_lib.py "10.0.0.250"
password1 client-intermediary

[+] Harversting beacon...
[+] Generating knock...
[+] Calculating port...
[+] Sending knock...
[+] Knock sent to 10.0.0.250 36688
```

```
root@kali:~# python crucible_server_lib.py

[+] Starting server...
[+] New beacon received.
[-] Checking beacon again in 4 minutes.
Restarting listener...
[-] Sniffing for UDP on port 36688.
[+] 128-Byte packet received from 10.0.0.248
[+] Authenticated knock received for command name client-intermediary
[+] Executing command: python crucible_client_lib.py "10.0.0.15" password2 intermediary-server

[+] Harversting beacon...
[+] Generating knock...
[+] Calculating port...
[+] Sending knock...
[+] Knock sent to 10.0.0.15 10323
[+] New beacon received.
Restarting listener...
[-] Checking beacon again in 4 minutes.
[-] Sniffing for UDP on port 29770.
^C
Exiting...
Finished.
```

```
root@kali2:~# python crucible_server_lib.py

[+] Starting server...
[+] New beacon received.
[-] Checking beacon again in 4 minutes.
Restarting listener...
[-] Sniffing for UDP on port 10323.
[+] 128-Byte packet received from 10.0.0.250
[+] Authenticated knock received for command name intermediary-server
[+] Executing command: echo "Routing knock received."
Routing knock received.
[-] Beacon unchanged, checking again in 30 seconds.
^C
Exiting...
Finished.
```

Fig. 6 Routing Crucible commands through intermediary hosts

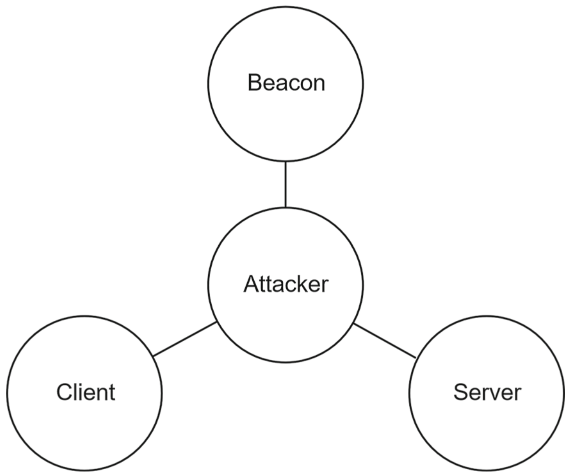


Fig. 7 Attacker positioning context

5.1.1 Attacks on identification protocols

Katz et al. [22] outlines a range of attacks that can be mounted against identification protocols, here they are reviewed in the context of Crucible:

- "impersonation": an attacker impersonating the client will only be able to authenticate against the server with use of the password. The beacon service is open-access, and would not suffer an attacker imitating the client or server. Impersonating the server, for example in a man-in-the-middle scenario, could grant an attacker access to knocks, and could prevent them reaching the legitimate server. From obtained knocks, there is arguably little an attacker could gain, as reversing the hash functions is intractable by design. If a knock value has already been issued, there is little to be gained from an attacker intercepting and relaying it themselves. Impersonation of the beacon, owing to HTTPS authentication via public key infrastructure, is assumed to be difficult.
- "replay attack": If a beacon value were replayed to the server, an attacker would have the opportunity to replay commands the client had already issued. If a beacon value was replayed to the client alone, the server would not accept that client's knocks. Both scenarios represent threats and illustrate the trust required in the beacon service, and its secure access. As the server is silent, only replay attacks seemingly sent from the client require further consideration, which are actively defended against per Fig. 3.
- "interleaving attack: an impersonation or other deception involving selective combination of information from one or more previous or simultaneously ongoing protocol executions." Interleaving attacks don't appear applicable here as the protocol has little interactivity. Further information on such attacks can be found in [60].
- " reflection attack: an interleaving attack involving sending information from an ongoing protocol execution back to the originator of such information." See interleaving attacks.
- "forced delay": without a beacon, neither client nor server can operate port knocking functions. Delaying port knocks from client to server could make them stale, and therefore unacceptable.
- "chosen-text attack: [...] an adversary strategically chooses challenges in an attempt to extract information about the claimant's long-term key." Impersonating the beacon, an attacker could try to send a message to recover the key through examination of the resulting knocks, though this would require subversion of the BLAKE2 hash function, which is both keyed, and invoked twice in the production of a knock value. It is unclear to the author how such an attack would be approached, and seemingly infeasible.

From the attacks discussed, most would require man-in-the-middle capabilities of an attacker. Impersonation of the beacon service for use in replay of a beacon, or chosen-beacon attacks, represent the only threats substantially different from an attacker with the capability to physically tamper with the network cable, causing delay or loss of service. This sort of impersonation could be carried out if an attacker had compromised the API's service, or if the requests made to the API via HTTPS were not sufficiently secured. If Bitcoin itself were manipulated, in order to influence beacon values, this should be considered in the same context as an attacker impersonating the beacon service. Bonneau et al. and Pierrot and Wesolowski [38], and [61] provide cost estimates for such an attack in the thousands of dollars. The chosen block outcome would further need to circumvent double-invocation of keyed BLAKE2.

5.1.2 Online attacks against the listening service

The `scapy` interface for lower-level `libpcap` functionality could provide adversaries an attack vector for Crucible. Indeed, the `libpcap` is not without historical vulnerabilities [62] and applications of `libpcap`, such as Wireshark, have experienced vulnerabilities as a result of this [63]. As noted previously, Crucible requires root permissions (for its raw packet access), and an exploitation of `libpcap` or `scapy` could have dire consequences. The listening service itself could also be vulnerable to a denial of service attack, as is the case for IDPS devices, where an attacker may send large volumes of traffic, sometimes anomalous in nature, " to attempt to exhaust a sensor's resources or cause it to crash" [64]. For Crucible, with its fail-closed stance, this would prevent client from authenticating until the service was restarted.

5.1.3 Offline attacks against passwords

In Crucible, the Argon2 hash is used as the key for BLAKE2, and this hash is retained on the server to authenticate clients. Should the server be compromised, and if its hash values are stolen and cracked (i.e., the passwords were discovered) an attacker could use these credentials to log into other client accounts, elsewhere. Password cracking is generally the process of inverting a given hash H_i, by discovering the password such that $H(password) = H_i$. This involves generating a great deal of passwords, hashing each, and comparing the results against the obtained hash to find a match.

If the password supplied to Argon2 is not of sufficient length, bruteforce attacks are made easier for an adversary, as there is less keyspace to search. Similarly, if the password is weakened by using predictable values (words, numbers, patterns etc.) then this makes dictionary attacks easier to mount. Further, the Argon2 hash is derived using a static salt value, meaning if future development added capability for multiple users, then the hashes derived from these passwords would be vulnerable to rainbow table attacks, whereby a single $H(password)$ can be tested against all of the server's user hashes. This could be solved by having the user remember a salt value (or username = salt), or by including a client state required for knocking, neither of which are preferable solutions. The key point, noted in 4.3.1, is that Argon's settings parametrise the difficulty for calculating a hash, making authentication *slightly* more slow for users, but dramatically increasing the cost to adversaries of mounting bruteforce or dictionary attacks.

5.1.4 Attacks against dependencies

Library and package dependencies are important in the context of secure protocol design, because external code vulnerabilities can result in exploitation of the prototype itself. As discussed earlier in Sect. 3.3.1, a NIST elliptic curve cryptographic standard was allegedly backdoored. Cimpanu [65] is a more recent example of a Python module for handling SSH connections, which surreptitiously harvested and exfiltrated the user's SSH credentials. Javascript library BrowseAloud was recently compromised, resulting in infection of websites using the library with cryptojacking software; repurposing their machines as cryptominers [66]. For transparency, Crucible's dependencies are included here in Table 2, and their use should be properly reviewed before deployment in a production environment. Once deployed, the libraries need to be constantly updated, and periodically checked against vulnerability databases.

5.1.5 Reconnaissance and stealth

Reconnaissance encompasses the methods an adversary can deploy in information gathering at the start of a campaign. The level of *stealth* a port knocking implementation provides may determine whether or not it is detected by an attacker conducting reconnaissance, and therefore stealth can decrease the likelihood that a port knocking server is exploited. An attacker performing reconnaissance activities could benefit from the following information, all of which could be made possible through detection of port knocking:

– Identification of a host as a client, server, or beacon service.
– Detection of a port knocking service on a server.
– Identification of the services hidden or protected by port knocking.
– Identification of the port knocking implementation, i.e., Crucible.

There are a number of characteristics that could indicate port knocking from captured traffic between client, server and beacon. Assuming an attacker had access to

Table 2 Dependencies in the Crucible prototype

	socket	getpass	Argon2	pyblake2	sys	Multiprocessing	Time	os	Scapy
Client	✓	✓	✓	✓	✓				
Server				✓	✓	✓	✓	✓	✓
3rd party?			✓	✓					✓

Crucible's documentation and code, *passive methods* of traffic analysis could look for indications such as:

– Periodic requests to the beacon service. These could be cross referenced with the beacon API to match a new block with a spike in HTTPS data transfer. If using the default API, an attacker could perform a reverse lookup to the API's URL and identify Crucible traffic this way.
– UDP packets with 512-bit payloads, where the destination port always differs. Other features of the knock packet may be identifiable, a number of techniques for passive fingerprinting of traffic are explored by [67]. Higher amounts of UDP traffic than expected may contradict stealth goals [68].
– Identifiable traffic resulting from a port knocking authorised command. For example, if the command was to open a port, an attacker could periodically diff open port scans of the server, and identify successful knocking traffic.

Sanai [69] explores *active methods* an attacker can use to identify use of promiscuous mode by a NIC on the local network, which is active when Crucible is running. One such method involves crafting an ARP packet, which would normally be rejected by the NIC, however in promiscuous mode the host issues no such reply. As seen in Fig. 8, the Nmap `sniffer-detect` (Nmap is a popular network sniffing tool) script performs this and other checks, and may allow an attacker to distinguish between Crucible and other port knocking solutions.

Greater discussion on stealth aspects of port knocking is reviewed in [6,70] and [71]. In comparison with other solutions examined in Sect. 2.1 on port knocking mechanics, Crucible has the following overall advantages and disadvantages in stealth:

– With only a single packet sent to the server for the knock, proportionally less traffic on the wire is attributable to Crucible than other solutions with more client-server interactivity.

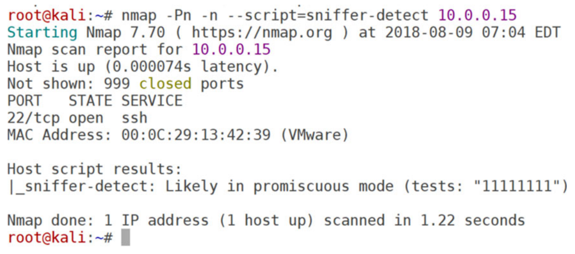

```
root@kali:~# nmap -Pn -n --script=sniffer-detect 10.0.0.15
Starting Nmap 7.70 ( https://nmap.org ) at 2018-08-09 07:04 EDT
Nmap scan report for 10.0.0.15
Host is up (0.000074s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
MAC Address: 00:0C:29:13:42:39 (VMware)

Host script results:
|_sniffer-detect: Likely in promiscuous mode (tests: "11111111")

Nmap done: 1 IP address (1 host up) scanned in 1.22 seconds
root@kali:~# 
```

Fig. 8 Indication of a Crucible server: promiscuous mode NIC

– No indication of the command executed on the server should be identifiable from the wire, as replay protection is enforced (no command will be re-issued), and as the command name is masked by keyed hashing. The definition of a keyed hash in Sect. 3.2.2 explains why the command is not recoverable from traffic. No client identifiers such as IP addresses or usernames are recoverable from network traffic, nor are service identifiers such as the service destination port. Very little information is leaked: between client and server, the only traffic exchanged is a single datagram with a pseudo-random number as its payload.
– A single UDP datagram containing a random number may look suspicious in a given context, Crucible makes no effort to deploy steganography, or other methods, to appear innocuous to an attacker.
– As a multi-party solution using a random beacon service, Crucible is more easily detected.

6 Conclusion

Zero knowledge proofs were introduced for private, lightweight client-identification. Chaos-based cryptography was explored for the purpose of minimalist, dependency-free cryptographic hashing. Random beacons were used to secure replay protection while preserving a single-packet knock, and preventing attacker-chosen computation. This paper has explored novel combinations of these topics with regards to port knocking and has used these ideas to develop Crucible.

Crucible achieves command authorisation using a single packet between client and server, with a payload likely indistinguishable from a random number. Crucible's design is minimalist, secure and stateless. It is portable, and the user only needs to memorise an IP, a password, and a command name in order to authenticate. Crucible does not authenticate post-knock traffic (see [11]) and a single command can only be executed once within the period of a random beacon, though an unlimited number of unique commands can be run in this window. Lastly, trust must be placed in the random beacon service, which is a non-trivial consideration.

Compliance with ethical standards

Ethical statement There are no potential conflicts of interest, and the research does not include human participants and/or animals. The work has been undertaken to accepted standards of ethics and of professional standards.

Appendix

Protocol 1 Schnorr Identification Protocol

Protocol setup

– Let p and q be two large primes where $p-1$ is a multiple of q. Let G_q denote the subgroup of $\mathbb{Z}_p^*$ of prime order q, and g be a generator for the subgroup.
– Let a be the private exponent chosen uniformly at random from $[0, q-1]$.Let $A = g^a \mod p$.
– Ahead of execution, both prover and verifier know (p, q, g, G, A).

Protocol actions

1. Prover chooses a number v uniformly at random from $[0, q-1]$ and computes $V = g^v \mod p$ and sends this to the Verifier.
2. Verifier chooses a challenge c uniformly at random from $[0, 2^{t-1}]$, where t is the bit length of the challenge, and sends this to the prover.
3. Prover computes $r = v - ac \mod q$ and sends this to the Verifier.
4. Verifier ensures:
   (a) A is within $[2, p-1]$
   (b) $A^q = 1 \mod p$
   (c) $V = g^r A^c \mod p$

5. *Protocol messages*

$$\text{Prover} \rightarrow \text{Verifier:} \quad V = g^v \mod p \tag{1}$$
$$\text{Prover} \leftarrow \text{Verifier:} \quad c \tag{2}$$
$$\text{Prover} \rightarrow \text{Verifier:} \quad r = v - ac \mod q \tag{3}$$

Notes
The protocol proves knowledge of the secret exponent a without revealing any information about it. Setup parameters may be chosen as per DSA choices. Checks 4(a) and (b) are to avoid invalid public keys. Check 4(c), since:
$g^r A^c = (g^{v-ac})(g^a)^c = g^{(v-ac+ac)} = g^v = V \mod p$. This protocol is referenced from [26], Section 2.2.

Protocol 2 Schnorr Non-interactive Zero-Knowledge Proof

Protocol setup

– Let p and q be two large primes where $p-1$ is a multiple of q. Let G_q denote the subgroup of $\mathbb{Z}_p^*$ of prime order q, and g be a generator for the subgroup.
– Let a be the private exponent chosen uniformly at random from $[0, q-1]$. Let $A = g^a \mod p$ be public key associated with a.
– Let H be a secure cryptographic hash function, *UserID* a unique identifier for the Prover, and *OtherInfo* optional data. The bit length of the hash output should be at least equal to that of the order q of the considered subgroup.

– Ahead of execution, both prover and verifier know $(p, q, g, G, A, \mathrm{H}, \mathrm{UserID})$.

Protocol actions

1. Prover chooses a number v uniformly at random from $[0, q-1]$ and computes the following:
   (a) $V = g^v \mod p$.
   (b) $c = \mathrm{H}(g \| V \| A \| \mathrm{UserID} \| \mathrm{OtherInfo})$
   (c) $r = v - ac \mod q$.
2. Prover sends $(\mathrm{UserID}, \mathrm{OtherInfo}, c, r)$ to Verifier.
3. Verifier uses the provided UserID to lookup A.
4. Verifier computes $V = g^r A^c$.
5. Verifier ensures $c \overset{?}{=} \mathrm{H}(g \| V \| A \| \mathrm{UserID} \| \mathrm{OtherInfo})$.

Protocol 3 Absolute Value Chaos-based Cryptographic Hash Function

Inputs

– A message string M of arbitrary length.
– A key K used as the initial value of the chaotic map. Chosen as a long floating number of 128-bit precision, and associated keyspace.
– A fixed number of map iterations i.
– A chosen range for the α coefficient to introduce chaos into the absolute value map, as seen in Equation (2).

Process

1. The message is padded with the ASCII character ’0’ ASCII appended to the suffix, until the message length in bytes is a multiple of 8.
2. The message M is split into an L-sized array of bytes ω_i for $i = 1 \ldots L$. Each byte element uses the ASCII integer value for the associated string character in M.
3. With key K substituting for ω_0 as the initialisation value, each ω_i is iterated through the chaotic map i-times: $x_{n+1} = 1 - ABS((0.0015\omega_m + 1.8)x_n)$
4. This is repeated for the next ω_{m+1}, using the previous x_i as the initialisation value ω_0, until each byte of the original message has been combined into x_L the last value.
5. The final value produced by the chaotic map x_L is normalised into a long number in the range $0 \leq H(M) < 2^{128}$ as the message digest.

Notes In step (3) the α value is equal to $0.0015\omega_i + 1.8$, this is because each ω_i is an integer ASCII value between 32 and 126, so these values are normalised onto the chosen range of $1.8 \leq \alpha < 2$ for the required chaotic behaviour. This protocol is referenced from [35] with modification to the α parameter.

Protocol 4 Bitcoin Random Beacon

Inputs

– Secure cryptographic hash function $H(m, k)$, here the chaos-based keyed hash function from earlier is used, with key k for message m.
– Block header B_H and block header hash B_{HH} of the Bitcoin network’s most recent block, polled from a Bitcoin tracking website (a blockchain API) by through a HTTPS request.

Process

1. Pull down the block header B_H and the hash of the block header B_{HH} from the website.
2. Calculate the binary OR of the header and the block header hash, $b = B_H + B_{HH}$
3. Output $H(b, k)$

Notes
This protocol is referenced from the implementation by [50] of the Bitcoin Random Beacon proposed in [38], with the following modifications:
Check is performed to validate the block header hash received from the website, this is to avoid importing libraries for calculating SHA hashes used in Bitcoin.
[50] chose a HMAC construction using SHA-256, instead the chaotic keyed hash from 3.2.3 is used. The block header hash is included to strengthen against malicious miners aiming to influence the block header. Even if the header were tampered with, the resulting hash value of this header remains unpredictable.

References

1. Sorrel-Dejerine, O.: “The spooky world of the ’numbers stations’—BBC News,” Apr (2014). [Online]. Available: https://www.bbc.co.uk/news/magazine-24910397

2. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: "Meltdown," *CoRR*, vol. abs/1801.01207 (2018). [Online]. Available: arXiv:1801.01207
3. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: "Spectre attacks: exploiting speculative execution," pp. 1–19 (2019)
4. Popeea, T., Olteanu, V., Gheorghe, L., Rughiniş, R.: Extension of a port knocking client-server architecture with NTP synchronization. In: 2011 RoEduNet International Conference 10th Edition: Networking in Education and Research, 6, pp. 1–5 (2011)
5. Srivastava, V., Keshri, A.K., Roy, A.D., Chaurasiya, V.K., Gupta, R.: "Advanced port knocking authentication scheme with QRC using AES." In: 2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC), 4, pp. 159–163 (2011)
6. Vasserman, E.Y., Hopper, N., Tyra, J.: Silentknock: practical, provably undetectable authentication. Int. J. Inf. Secur. **8**(2), 121–135 (2009). https://doi.org/10.1007/s10207-008-0070-1
7. Lunsford, P., Wright, E.C.: Closed port authentication with port knocking. Age **10**, 1 (2005)
8. Bou-Harb, E., Debbabi, M., Assi, C.: Cyber scanning: a comprehensive survey. IEEE Commun. Surv. Tutor. **16**(3), 1496–1519 (2014)
9. deGraaf, R., Aycock, J., Jacobson, M.: "Improved port knocking with strong authentication." In: 21st Annual Computer Security Applications Conference (ACSAC'05), 12 (2005)
10. Jha, S.: An object oriented approach for port knocking. IJNIET **6(1) (2016)**
11. Jeanquier, S.: An Analysis of Port Knocking and Single Packet Authorization. University of London, Royal Holloway (2016)
12. Sel, D.: Authenticated Scalable Port-Knocking. Technical University of Munich, Munich (2016)
13. Sel, D., Totakura, S.H., Carle, G.: "sKnock: port-knocking for masses." In: 2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW). IEEE 1–6 (2016)
14. Tiwari, R.: Port–knocking system using unilateral authentication algorithm. 9 (2013)
15. Al-Bahadili, H., Hadi, A.H.: Network security using hybrid port knocking. IJCSNS **10**(8), 8 (2010)
16. Liew, J.-H., Lee, S., Ong, I., Lee, H.-J., Lim, H.: "One-time knocking framework using SPA and IPsec." In: 2010 2nd International Conference on Education Technology and Computer, vol. 5, 6 (2010)
17. Worth, D.: COK: cryptographic one-time knocking. (2004)
18. Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. Wiley, Hoboken (2007)
19. Mohr, A.: A survey of zero-knowledge proofs with applications to cryptography. Southern Illinois University, Carbondale, pp. 1–12 (2007)
20. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press, Cambridge (2009)
21. Mollin, R.A.: Cryptography and zero knowledge. Int. J. Pure Appl. Math. **31**(3), 345–360 (2006)
22. Katz, J., Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
23. Goldreich, O.: Modern Cryptography, Probabilistic Proofs and Pseudorandomness, 17th edn. Cambridge University Press, Cambridge (1998)
24. Mao, W.: Modern Cryptography: Theory and Practice. Prentice Hall PTR, Upper Saddle River (2003)
25. Van Tilborg, H .C., Jajodia, S.: Encyclopedia of Cryptography and Security. Springer, Berlin (2014)
26. Hao, F., Metere, R., Shahandashti, S.F., Dong, C.: Analyzing and patching speke in iso/iec. IEEE Trans. Inf. Forensics Secur. **13**(11), 2844–2855 (2018)
27. Alvarez, G., Li, S.: Some basic cryptographic requirements for chaos-based cryptosystems. Int. J. Bifurc. Chaos **16**(08), 2129–2151 (2006)
28. Kanso, A., Ghebleh, M.: A fast and efficient chaos-based keyed hash function. Commun. Nonlinear Sci. Numer. Simul. **18**(1), 109–123 (2013)
29. Kocarev, L.: Chaos-based cryptography: a brief overview. IEEE Circuits Syst. Mag. **1**(3), 6–21 (2001)
30. Li, C., Lin, D., Lü, J., Hao, F.: Cryptanalyzing an image encryption algorithm based on autoblocking and electrocardiography. IEEE MultiMed. **25**(4), 46–56 (2018)
31. Zhen, P., Zhao, G., Min, L., Li, X.: "A survey of chaos-based cryptography." In: 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, vol. 11, pp. 237–244 (2014)
32. Teh, J.S., Samsudin, A., Akhavan, A.: Parallel chaotic hash function based on the shuffle-exchange network. Nonlinear Dyn. **81**(3), 1067–1079 (2015). https://doi.org/10.1007/s11071-015-2049-6
33. Li, C., Zhang, Y., Xie, E.Y.: When an attacker meets a cipher-image in 2018: a year in review. J. Inf. Secur. Appl. **48**, 102361 (2019)
34. Aumasson, J.: Serious Cryptography: A Practical Introduction to Modern Encryption. No Starch Press, San Francisco (2017)
35. San-Um, W., Srichavengsup, W.: "A topologically simple keyed hash function using a single robust absolute-value chaotic map." In: 2013 IEEE International Conference on Communication, Networks and Satellite (COMNETSAT), vol. 12, pp. 95–99 (2013)
36. Luo, Y., Liu, Y., Liu, J., Ouyang, X., Cao, Y., Ding, X.: Ecm-ibs: a chebyshev map-based broadcast authentication for wireless sensor networks. Int. J. Bifurc. Chaos **29**(09), 1950118 (2019)
37. Rabin, M.O.: Transaction protection by beacons. J. Comput. Syst. Sci. **27**(2), 256–267 (1983)
38. Bonneau, J., Clark, J., Goldfeder, S.: "On Bitcoin as a public randomness source." Cryptology ePrint Archive, Report 2015/1015 (2015). [Online]. Available: https://eprint.iacr.org/2015/1015
39. Jiwa, A., Seberry, J., Zheng, Y.: Beacon based authentication. In: Gollmann, D. (ed.) Computer Security—ESORICS 94, pp. 123–141. Springer, Berlin (1994)
40. Clark, J., Hengartner, U.: "On the use of financial data as a random beacon." Cryptology ePrint Archive, Report 2010/361 (2010). [Online]. Available: https://eprint.iacr.org/2010/361

41. Lee, H.H., Chang, E.-C., Chan, M.C.: Pervasive random beacon in the internet for covert coordination. In: Barni, M., Herrera-Joancomartí, J., Katzenbeisser, S., Pérez-González, F. (eds.) Information Hiding, pp. 53–61. Springer, Berlin (2005)
42. Lenstra, A. K., Wesolowski, B.: Trust, and public entropy: a unicorn hunt. (2016)
43. Bennett, C.H., Smolin, J.A.: “Trust enhancement by multiple random beacons.” *CoRR*, vol. cs.CR/0201003 (2002). [Online]. Available: arXiv:cs.CR/0201003
44. Ferguson, N., Schneier, B.: Practical Cryptography, vol. 23. Wiley, New York (2003)
45. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: “Verifiable delay functions.” Cryptology ePrint Archive, Report 2018/601 (2018). [Online]. Available: https://eprint.iacr.org/2018/601
46. Barski, C., Wilmer, C.: Bitcoin for the Befuddled. No Starch Press, San Francisco (2014)
47. Mehran, P., Reza, E. A., Laleh, B.: “SPKT: secure port knock-tunneling, an enhanced port security authentication mechanism.” In: 2012 IEEE Symposium on Computers Informatics (ISCI), 3, pp. 145–149 (2012)
48. Mahbooba, B., Schukat, M.: “Digital certificate-based port knocking for connected embedded systems.” In: 2017 28th Irish Signals and Systems Conference (ISSC), 6, pp. 1–5 (2017)
49. Koch, W., Bestavros, A.: “PROVIDE: hiding from automated network scans with proofs of identity.” In: 2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), 10, pp. 66–71 (2016)
50. Lee, C.: “Bitcoin beacon: using the blockchain to generate random bits.” 7 (2017). [Online]. Available: http://charlesjlee.com/post/20170716-bitcoin-beacon/
51. BTC.com.: “API Documentation—BTC.com.” (2018). [Online]. Available: https://btc.com/api-doc
52. Wetzels, J.: “Open sesame: the password hashing competition and Argon2.” Cryptology ePrint Archive, Report 2016/104 (2016). [Online]. Available: https://eprint.iacr.org/2016/104
53. Biryukov, A., Dinu, D., Khovratovich, D., Josefsson, S.: “The memory-hard Argon2 password hash and proof-of-work function.” Working Draft, IETF Secretariat, Internet-Draft draft-irtf-cfrg-argon2-03, 8 (2017). [Online]. Available: http://www.ietf.org/internet-drafts/draft-irtf-cfrg-argon2-03.txt
54. Biryukov, A., Dinu, D., Khovratovich, D.: “Argon2: new generation of memory-hard functions for password hashing and other applications.” In: 2016 IEEE European Symposium on Security and Privacy (EuroS P), 3, pp. 292–302 (2016)
55. Saarinen, M.-J., Aumasson, J.: The BLAKE2 cryptographic hash and message authentication code (MAC). Internet Requests for Comments RFC Editor RFC 7693, 11 (2015)
56. Sokolovskiy, A.: “pyblake2—BLAKE2 hash function for Python.” (2013). [Online]. Available: https://pythonhosted.org/pyblake2/
57. Aumasson, J., Neves, S., Wilcox-O’Hearn, Z., Winnerlein, C.: “Fast secure hashing.” (2017). [Online]. Available: https://blake2.net/
58. Postel, J.: “Internet control message protocol.” Internet requests for comments, RFC Editor, STD 5, 9 (1981). [Online]. Available: http://www.rfc-editor.org/rfc/rfc792.txt
59. Kirsch, J.: “Knock: Practical and secure stealthy servers,” (2014). [Online]. Available: https://gnunet.org/
60. Bird, R., Gopal, I., Herzberg, A., Janson, P., Kutten, S., Molva, R., Yung, M.: Systematic design of two-party authentication protocols. In: Feigenbaum, J. (ed.) Advances in Cryptology—CRYPTO ’91, pp. 44–61. Springer, Berlin (1992)
61. Pierrot, C., Wesolowski, B.: Malleability of the blockchain’s entropy. Cryptology ePrint Archive, Report 2016/370, (2016). [Online]. Available: https://eprint.iacr.org/2016/370
62. “CVE-2011-1935,” Available from MITRE, CVE-ID CVE-2011–1935 (2011). [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2011-1935
63. “CVE-2014-4174, ”Available from MITRE, CVE-ID CVE-2014–4174 (2014). [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2014-4174
64. Scarfone, K., Mell, P.: Guide to intrusion detection and prevention systems (idps). NIST Spec. Publ. **800**(2007), 94 (2007)
65. Cimpanu, C.: Backdoored Python Library Caught Stealing SSH Credentials 5 (2018). [Online]. Available: https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/
66. NCSC, NCSC advice: malicious software used to illegally mine cryptocurrency 2 (2018). [Online]. Available: https://www.ncsc.gov.uk/guidance/ncsc-advice-malicious-software-used-illegally-mine-cryptocurrency
67. Zalewski, M.: Silence on the Wire: a Field Guide to Passive Reconnaissance and Indirect Attacks. No Starch Press, San Francisco (2005)
68. Manzanares, A .I., Márquez, J .T., Estevez-Tapiador, J .M., Castro, J .C .H.: Attacks on port knocking authentication mechanism. In: Gervasi, O., Gavrilova, M .L., Kumar, V., Laganá, A., Lee, H .P., Mun, Y., Taniar, D., Tan, C .J .K. (eds.) Computational Science and Its Applications—ICCSA 2005, pp. 1292–1300. Springer, Berlin (2005)
69. Sanai, D.: Detection of Promiscuous Nodes Using ARP Packets (2001). [Online]. Available: www.securityfriday.com/promiscuous_detection_01.pdf
70. Mileva, A., Panajotov, B.: Covert channels in TCP/IP protocol stack. Cent. Eur. J. Comput. Sci. **4**(2), 45–66 (2014). https://doi.org/10.2478/s13537-014-0205-6
71. Wendzel, S., Keller, J.: Hidden and under control. Ann. Telecommun. Ann. des Télécommun. **69**(7), 417–430 (2014). https://doi.org/10.1007/s12243-014-0423-x