Autenticação

Autenticação: processo de verificação de uma identidade alegada por ou para uma entidade do sistema

1. **Identificação:** especificar identificador
2. **Verificação:** vincular entidade (pessoa) a identificador

Meios de Autenticação
1. Conhece
2. Possui
3. É – biometria estática
4. Faz – biometria dinâmica

→ **Palavra-Passe:** popular por custo e conveniência

Vulnerabilidades
1. dicionário offline
2. conta específica
3. palavra-passe popular
4. adivinhação de palavra-passe
5. roubo de máquina
6. erros do utilizador
7. uso de múltiplas palavras-passe
8. monitorização eletrónica

Mitigações
1. impedir acesso não autorizado a ficheiro de palavras-passe
2. deteção de intrusões
3. bloqueio de conta
4. políticas
5. formação
6. fim de sessão automático
7. encriptação de ligações de rede

· O ficheiro de palavras-passe só deve conter as "hashes"

Ataque de Dicionário: pré-computar $h(x)$ para todo o x num dicionário de palavras-passe comuns

↓ SOLUÇÃO

Sal: aleatório não secreto — $y = h(p, s) \rightarrow (s, y)$

Ataque de "Rainbow Table": pré-computar tabelas de valores de hash para todos os sais

↳ SOLUÇÃO
↳ SAL MAIOR!

Chave Criptográfica: 64 bits → 2^{64} chaves → aleatório → 2^{63} tentativas
Palavra-Passe: 8 caracteres → $256^8 = 2^{64}$ palavras-passe → não aleatório → $<<< 2^{63}$

· A palavra-passe recomendada é uma frase-passe!

Ataque: Externo → Utilizador → Administrador

· O acesso ao ficheiro local de palavras-passe cifradas deve ser negado para bloquear ataques de adivinhação offline — só permitido para utilizadores privilegiados e usando um ficheiro "sombra"

Técnicas: educação; geração; verificação reativa; verificação proativa

Verificação Proativa:
1. Regras e Conselhos
2. Cracker
3. Modelo de Markov: gera palavras-passe fáceis para rejeitar
4. Filtro de Bloom: constrói tabela baseada em dicionário

Biometria

Autenticação Biométrica: com base nas características físicas do utilizador

Biometria Ideal
1. **Universal:** aplica-se a (quase) todos
2. **Distinguível:** distingue-se com certeza
3. **Permanente:** característica física medida nunca muda
4. **Colecionável:** fácil de recolher dados necessários

Identificação: "quem vai lá?" 1-* sujeitos não cooperativos
Autenticação: "tu és quem tu dizes ser?" 1-1 sujeitos cooperativos

Fraude: T autenticado como A — Falso Positivo
Insulto: A não autenticado como A — Falso Negativo
↺ INVERSOS

Registo: a informação biométrica do sujeito é colocada na base de dados
↳ medição cuidadosa que pode ser lenta e repetida, mas tem de ser precisa

Reconhecimento: deteção biométrica, quando usada na prática
↳ tem de ser rápido, simples e razoavelmente exato

Impressão Digital: pontos extraídos são comparados com a informação armazenada na base de dados usando correspondências estatísticas

Geometria da Mão: medição da forma da mão
☺ rápido; mãos simétricas
☹ inutilizável em jovens ou idosos; rácio de erros relativamente igual

Padrões da Íris: desenvolvimento caótico com pouca influência genética, mas padrão estável

Tokens

Token: objeto que o utilizador possui para se autenticar — difícil de copiar / bem-protegido

Cartão de Memória: armazena dados, mas não processa
∴ leitor especial; perda; insatisfação

SmartCard: tem o seu próprio processador, memória (ROM, EEPROM e RAM) e portos de I/O — executa protocolo para autenticar leitor/computador

Problemas com Palavras-Passe:
1. Seleção: palavras-passe boas e seguras são difíceis de encontar
2. Memorização: fácil de esquecer & difícil de lembrar
3. Reutilização: demasiadas palavras-passe para memorizar
4. Partilha: não-repúdio é fácil de contornar com partilha de palavras-passe
5. Malware: "sniffing"; "phishing"; ...

Segundo Fator Clássico de Autenticação: token físico de propósito único

Segundo Fator Flexível de Autenticação:
1. YubiKey: gerador de tokens "one time password" (OTP) de baixo custo
2. Telemóvel
   a. SMS: caro; lento; complicado; inseguro
   b. Aplicação: segurança baseada na sincronização temporal e segredo inicial

QR-Login: rápido; conveniente; segredo não memorizado → longo e complexo; resistente a vírus/KeyLogger/phishing → seguro em computadores não confiáveis; palavra-passe aleatoriamente gerada; fácil; encriptado; não interetável; privado

FIDO: especificações + interoperabilidade + licenciamento → AUTENTICAÇÃO

FIDO

Cliente FIDO: interage com FIDO Authenticators usando a camada de abstração através da API e com um agente de utilizador no dispositivo via interfaces

Servidor FIDO: interage com o servidor web confiável para comunicar mensagens protocolares, valida as atestações do autenticador contra os metadados, gere a associação de autenticadores a utilizadores e avalia/confirma as respostas a transações

Protocolos FIDO: trocam mensagens entre dispositivos do utilizador e partes confiáveis
1. Registo de Autenticadores: descobre os disponíveis, verifica-os e regista-os
2. Autenticação de Utilizadores: protocolo criptográfico desafio-resposta
3. Confirmação de Transações Seguras: apresentação de mensagem para confirmação
4. Desregisto de Autenticadores: quando o utilizador é removido da parte confiável

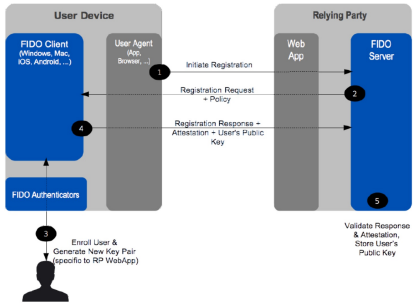
- A camada de abstração fornece uma API para autenticação de clientes

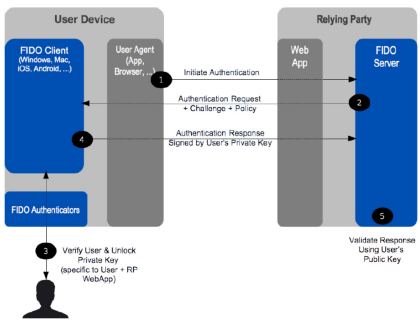
Autenticador: entidade segura conectada ou hospedada no dispositivo do utilizador que pode criar material chave associado a uma parte confiável — atesta-se

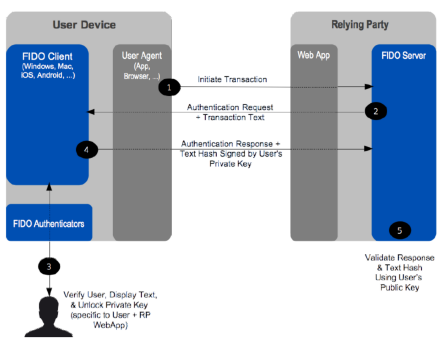
Atestação: como um Autenticador alega a uma parte confiável durante o registo que as chaves que gera e/ou certas medições que reporta originam de dispositivos genuínos com características certificadas — assinatura validada

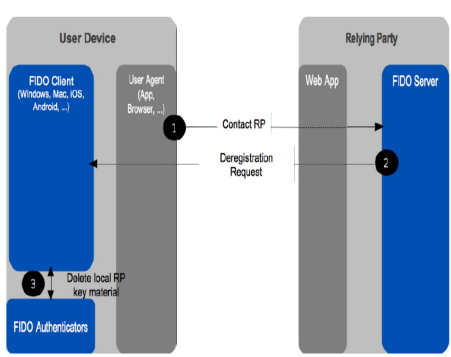
- Os utilizadores adquirem Autenticadores e registam-nos UNLINKABILITY
- Um dispositivo UAF não tem um identificador global a uma ou várias partes
- O protocolo UAF gera chaves por dispositivo, conta e parte com mínimos dados
- A verificação UAF é local, em dispositivos explicitamente aprovados

→ U2F → FIDO 2


User Device
Relying Party
FIDO Client
(Windows, Mac, iOS, Android, ...)
User Agent
(App, Browser, ...)
Web App
FIDO Server
1
Initiate Registration
2
Registration Request
+ Policy
4
Registration Response +
Attestation + User's Public Key
5
Validate Response
& Attestation,
Store User's
Public Key
FIDO Authenticators
3
Enroll User &
Generate New Key Pair
(specific to RP WebApp)


User Device
Relying Party
FIDO Client
(Windows, Mac, iOS, Android, ...)
User Agent
(App, Browser, ...)
Web App
FIDO Server
1
Initiate Authentication
2
Authentication Request
+ Challenge + Policy
4
Authentication Response
Signed by User's Private Key
5
Validate Response
Using User's
Public Key
FIDO Authenticators
3
Verify User & Unlock
Private Key
(specific to User + RP
WebApp)


User Device
Relying Party
FIDO Client
(Windows, Mac, iOS, Android, ...)
User Agent
(App, Browser, ...)
Web App
FIDO Server
1
Initiate Transaction
2
Authentication Request
+ Transaction Text
4
Authentication Response +
Text Hash Signed by User's
Private Key
5
Validate Response
& Text Hash
Using User's
Public Key
FIDO Authenticators
3
Verify User, Display Text,
& Unlock Private Key
(specific to User + RP
WebApp)


User Device
Relying Party
FIDO Client
(Windows, Mac, iOS, Android, ...)
User Agent
(App, Browser, ...)
Web App
FIDO Server
1
Contact RP
2
Deregistration
Request
3
Delete local RP
key material
FIDO Authenticators

Controlo de Acessos

Controlo de Acesso: prevenção do uso não autorizado de um recurso, incluindo a prevenção do uso de um recurso de forma não autorizada

Autenticação: restrições em quem (ou o quê) pode aceder ao sistema
→ tu és quem tu dizes que és?

Autorização: restrições nas ações dos utilizadores autenticados
→ tu tens permissão para fazer isso?

Sujeito: entidade que pode aceder a objetos

Objeto: recurso com acesso controlado

Direito de Acesso: forma como um sujeito acede a um objeto

Controlo de Acessos Discricionário: matriz esparsa de controlo de acessos que lista sujeitos e objetos em que cada entrada especifica os direitos de acesso do sujeito especificado a esse objeto

Lista de Controlo de Acesso: matriz por coluna — para cada objeto/recurso
☺ orientado aos dados; fácil de gerir recursos

Capacidades: matriz por linha — para cada sujeito/utilizador
☺ fácil de delegar; fácil de gerir utilizadores
☹ difícil de implementar

Deputado Confuso: separação da autoridade do propósito para que é usada
Ex: compilador a atuar em nome de outrem

Segurança Multi-Nível

UNIX: as permissões dos grupos estão nos ficheiros — OWNER RWX GROUP RWX OTHERS RWX
SetID: o sistema usa temporariamente os direitos do proprietário/grupo para permitir que programas privilegiados acedam a recursos/ficheiros normalmente inacessíveis
Sticky Bit: limita renomear, mover ou apagar diretórios ao proprietário
Superuser: está isento das restrições de controlo de acesso usuais

Classificação: aplica-se a objetos "Clearance": aplica-se a sujeitos
· O nível de segurança denota-se L(O) e L(S)
Segurança Multi-Nível (MLS): forma de controlo de acesso num sistema

Bell-LaPadula: lida com confidencialidade para prevenir leituras não autorizadas

Condição de Segurança Simples: S pode ler O se e só se $L(O) \ll L(S)$ NO READ UP
Propriedade Estrela (*): S pode escrever O se e só se $L(S) \ll L(O)$ NO WRITE DOWN
Tranquilidade Forte: etiquetas de segurança nunca mudam
Tranquilidade Fraca: etiquetas de segurança só mudam se "não violarem política"

Canal Coberto: caminho de comunicação não previsto como tal pelos projetistas do sistema
Ex: criar um ficheiro sinaliza 1; remover o ficheiro significa 0

1. O emissor e o recetor têm um recurso partilhado
2. O emissor consegue variar uma propriedade do recurso que o recetor pode observar
3. A comunicação entre o emissor e o recetor pode ser sincronizada

¡É virtualmente impossível eliminar canais cobertos em qualquer sistema útil!
↳ A capacidade dos canais cobertos deve ser reduzida para 1 bit/segundo

Ex: esconder dados nos campos do TCP — reservado OU número de sequência

Controlo de Inferências

- Respostas a questões gerais podem libertar informações específicas!
- Remover nomes não é suficiente!

↓ SOLUÇÃO

1. Não retornar uma resposta se o tamanho do conjunto for demasiado pequeno
2. Não libertar estatísticas se K% ou mais é devido a N ou menos
3. Adicionar pequenas quantidades de ruído aleatório aos dados

CAPTCHA

Completely Automated Public Turing test to tell Computers and Humans Apart

Teste de Turing: um humano questiona um humano e um computador, sem ver ambos — se o inquisidor não conseguir distinguir o humano do computador, o computador passa o teste

Automatizado: teste gerado e avaliado por um programa de computador
Público: programa e dados públicos
Teste de Turing: humanos conseguem passar o teste, mas máquinas não

- O computador cria e avalia testes que não consegue passar!
- O CAPTCHA é usado para <u>controlo de acesso</u>!

CAPTCHA
- <u>Visual</u>: problema de Optical Character Recognition
- <u>Áudio</u>: palavras distorcidas ou música

- OCR é um problema desafiante de IA — problema de segmentação

Software e Segurança

- Toda a segurança de informação está implementada em software!
- O software é uma fundação fraca para segurança!

→ "A complexidade é a inimiga da segurança" 5 bugs / 10000 LOC

Skynet: tolerância à introdução de falhas — degradação sob intrusão, mas correção

Erro no Programa: engano de programação — não intencional
Defeito: estado incorreto interno do programa — não observável externamente
Falha: comportamento diferente do esperado — observável externamente

→ Buffer Overflow

- O adversário pode injetar código sem saber (1) o endereço do código mau e (2) a localização do endereço de retorno na stack

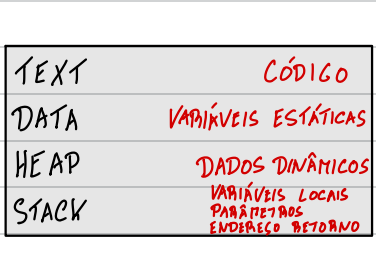
↓ PARA ISSO

- O adversário pode (1) preceder o código mau com NOP e (2) inserir muitos novos endereços de retorno

Prevenção: 1. stack não executável 2. linguagens seguras 3. funções seguras 4. canário

→ Mediação Incompleta: falha de validação do input

→ "Race Condition": alteração entre estados de processos críticos que deviam ser atómicos
Ex: mkdir



Malware

Vírus: propagação passiva
Worm: propagação ativa
Cavalo de Troia: funcionalidade inesperada
Trapdoor/BackDoor: acesso não autorizado
Coelho: exaurir os recursos do sistema

· Os vírus podem viver no setor de inicialização, em memória, aplicações, dados, rotinas, compiladores, debuggers, firmware, ...

1. Vírus Brain: instalado no setor de inicialização, evitava deteção
2. Worm Morris: carregador de inicialização para código encriptado que compilava, executava disfarçadamente e se apagava quando detetado

WORMS
3. Code Red: exploração de buffer overflow para se espalhar e causar DoS
4. SQL Slammer: cabia no limite de pacotes pequenos (376 bytes) da firewall

DETEÇÃO DE MALWARE

Deteção de Assinatura: pesquisa por string de bits no software
☺ eficaz em malware tradicional; esforço mínimo para utilizadores/administradores
☹ tamanho grande; pesquisa lenta; necessidade de atualização; incapaz para desconhecidos

Deteção de Alterações: se um valor de hash de um ficheiro muda, pode estar infetado
☺ sem falsos negativos; capaz para desconhecidos
☹ muitos falsos positivos; muito esforço para utilizadores/administradores

Deteção de Anomalias: monitorização do sistema por algo unusual ou suspeito
☺ capaz para desconhecidos
☹ não provado na prática; o anormal pode parecer normal

Futuro do Malware

Malware Polimórfico: worm polimórfica enciptada com uma nova chave de cada vez que se propaga — detetável por assinatura do código desenciptado

Malware Metamórfico: worm que muta antes de infetar um novo sistema para evitar deteções por assinatura — desmonta-se, varia-se e arranja-se, monta-se

Worm de Warhol: 15 MIN worm semente com uma lista inicial com um conjunto de endereços IP vulneráveis e cada infeção inicial ataca uma parte do espaço de endereços IP

Worm Flash: 15 SEG pré-determina todos os endereços IP vulneráveis e embebê-los na worm para, quando replicas, dividi o espaço de endereços vulnerável — IDS pessoal

Briareos: framework modular para deteção e prevenção de intrusões

```
    program V :=

{goto main;
    1234567;

    subroutine infect-executable :=
        {loop:
        file := get-random-executable-file;
        if (first-line-of-file = 1234567)
          then goto loop
          else prepend V to file; }

    subroutine do-damage :=
        {whatever damage is to be done}

    subroutine trigger-pulled :=
        {return true if some condition holds}

main:   main-program :=
        {infect-executable;
        if trigger-pulled then do-damage;
        goto next;}

next:

}
```

Vírus: pedaço de software que infeta programas — específico para cada sistema
Fases: 1. Dormente 2. Propagação 3. Ativação 4. Execução

Componentes:
1. Mecanismo de Infeção: permite replicação
2. Gatilho: evento que torna o payload ativo
3. Payload: o que faz, maligno ou benigno

Macro Vírus: explora as capacidades de macros das aplicações Office
E-mail Vírus: envia e-mail para toda a lista de endereços e causa dano local

Medidas: 1. Prevenção 2. Deteção 3. Identificação 4. Remoção
Anti-Vírus: 1. Assinaturas 2. Heurísticas 3. Ações 4. Combinação

Desencriptação Genérica (GD): corre os ficheiros executáveis através de um scanner GD que emula o CPU para interpretar instruções e verificar as assinaturas dos vírus — deixa o vírus desencriptar-se a si próprio

Worm: programa replicante que se propaga na rede — pode disfarçar-se
Fases: 1. Dormente 2. Propagação (pesquisa por outros sistemas, conecta-se, copia-se e executa) 3. Ativação 4. Execução
Medidas: AV & NIDS

Bot: programa que toma o controlo de outros computadores para lançar ataques difíceis de detetar — facilidade de controlo remoto e mecanismo de disseminação
Rootkit: conjunto de programas instalado para acesso de administrador — pode esconder a sua existência, ser persistente ou em memória e em modo utilizador ou Kernel

Ataque Salame: programador "fatia"/extrai bits valiosos — "insiders"
Ataque de Linearização: ataque de timing por canais colaterais — eficiente
Bomba Temporal: destruição de dados após um determinado período de tempo

Engenharia Reversa de Software

Engenharia Reversa de Software/Código: assume-se que o engenheiro reverso é um atacante que só tem um executável e quer compreendê-lo/modificá-lo

Disassembler: converte o executável para "assembly" — estático
Debugger: para através do código para o compreender completamente — dinâmico
Editor Hexadecimal: faz alterações a um ficheiro executável

Ex: AND eax, eax → XOR eax, eax

Técnicas Anti-Disassembly: confundir vista estática do código
↳ encriptação; falsificação; auto-modificação

Técnicas Anti-Debugging: confundir vista dinâmica do código
↳ monitorização de registos e breakpoints; threads

Resistência a Fraude: código verificar-se a si próprio para detetar fraude
↳ hash

Ofuscação de Código: tornar código mais difícil de compreender
↳ código esparguete; código morto; predicados opacos

→ "Break Once, Break Everywhere" (BOBE)

Software Clonado: o mesmo ataque funciona contra todas as cópias do software

Software Metamórfico: instâncias únicas, funcionalmente idênticas, mas com estrutura interna diferente — melhor opção contra BOBE

Gestão de Direitos Digitais

Gestão de Direitos Digitais: problema de controlo remoto — distribuir conteúdo digital e reter algum controlo sobre o seu uso, após a entrega

Proteção Persistente: como forçar restrições no uso de conteúdo após a entrega?
Ex: não copiar/encaminhar; limitar leituras/tempo

- A criptografia é necessária para entregar os bits e prevenir ataques triviais, mas não é a solução
- O estado atual é segurança por obscuridade, em violação do Princípio de Kerckhoffs
- A segurança por software é impossível devido a engenharia reversa

Buraco Analógico: o conteúdo mostrado pode sempre ser captado de forma analógica

Servidor: aplica a proteção persistente desejada e autentica o cliente SDS
Cliente: autentica-se perante o servidor e pede-lhe a chave criptográfica PLUGIN

Segurança: camada de resistência a fraude aplicada sobre código ofuscado (CÓDIGO ENCRIPTADO & ANTI-DEBUGGER) + hash; anti-captura de ecrã; marca de água; metamorfismo/individualização (SEGREDOS CRIPTOGRÁFICOS)

Ataques: spoofing; Man-in-the-Middle; repetição/redistribuição; captura do texto

Algoritmo de "Scrambling": algoritmo semelhante a enciptação — metamorfismo forte
1. Negociação do Algoritmo
2. Desenciptação no recetor para remover a enciptação forte
3. "De-scrambling" no driver do dispositivo antes de "rendering"

Servidor: tem muitos algoritmos de "scrambling"
Cliente: tem um subconjunto de algoritmos, enciptado com a chave do servidor

DADOS ⇄ DADOS "SCRAMBLED" ⇄ DADOS "SCRAMBLED" ENCRIPTADOS

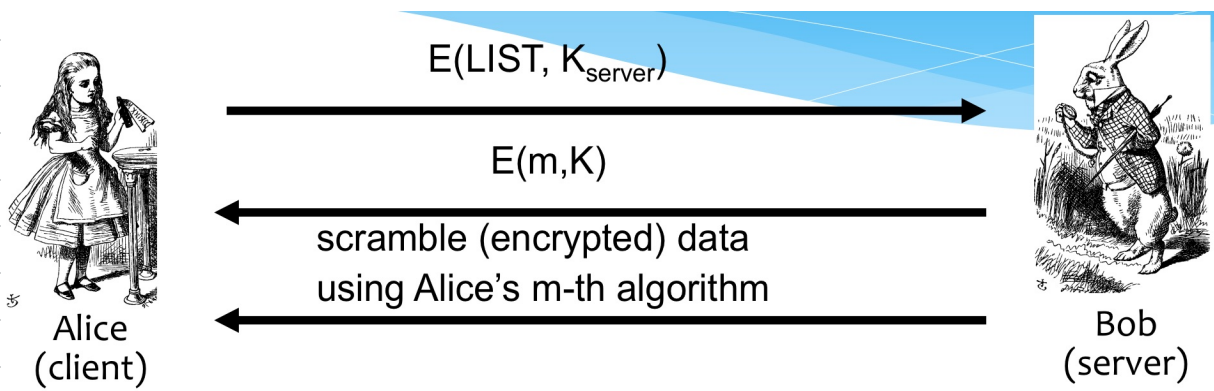
· O objetivo é manter o texto original longe de um potencial atacante, pela capacidade aplicar o algoritmo de "scrambling" embutido no driver do dispositivo no último momento

Porquê "Scrambling"? O vendor só escolhe algoritmos seguros e proprietários
Porquê Metamorfismo? A segurança por obscuridade ajuda contra engenharia reversa

"Peer Offering Service" (POS): o conteúdo legal protegido pode ser descarregado através do pagamento de uma pequena taxa

· As proteções como DRM são necessárias para conformidade regulamentar!

Problemas: gestão de políticas; autenticação



DESENVOLVIMENTO DE SOFTWARE SEGURO

"Penetrate and Patch": abordagem usual, mas péssima! FALÁCIA

Identidade Digital

Identidade: conjunto de atributos distintos — concessão individual do próprio
Direito: posse ou controlo sobre um determinado conjunto de recursos
Autenticação: apresentação de provas de identidade
Credencial/Reivindicação: prova para assumir identidade — contém atestações
↳ permite identificar e autenticar características e relações entre entidades

Leis da Identidade: princípios a reger por um sistema de identidade
1. Controlo e Consentimento do Utilizador: só revelar informações com consentimento
2. Divulgação Mínima para Uso Restrito: revelar o mínimo de informações
3. Partes Justificáveis: limitar divulgação de informações às partes necessárias/justificáveis
4. Identidade Direcionada: identificadores omnidirecionais (públicos) e unidirecionais (privados)
5. Pluralismo de Operadores e Tecnologias: interoperabilidade de tecnologias
6. Integração Humana: humano como componente do sistema distribuído
7. Experiência Consistente em Diferentes Contextos: experiência simples e consistente, mas separação de contextos

Identidade Digital: conjunto de atributos que descrevem de forma única um sujeito ou objeto digital e as suas relações com outros — resultante da desmaterialização
Identificador: atributo usado para reconhecer inequivocamente uma entidade num determinado contexto e para um determinado fim

Gestão da Identidade Digital: gestão do ciclo de vida e disponibilização de atributos de identidade de uma entidade a aplicações e serviços

Identity Manager: serviço responsável pela gestão de identidade digital
Identity Provider: fornece atributos de identidade para um utilizador de serviços
Fonte Oficial: entidade que possui o poder último de associar atributos a entidades
Alegação/Reivindicação: declaração que afeta a validade da associação de atributo
Credencial: agrupamento de uma ou mais alegações/reivindicações afirmadas/atestadas

MODELOS DE GESTÃO DE IDENTIDADE

Centralizado: o utilizador estabelece uma identidade com uma organização criando uma conta num site ou aplicação
∴ esforço do utilizador; diferentes políticas; não portabilidade; efeito "honeypot"

Agregado: colocar um Identity Provider a intermediar a relação entre o utilizador e o serviço providenciado por uma ou mais organizações — Autenticação.GOV
∴ intermediação do IdP — não portabilidade e efeito "honeypot"; não "offline"

Federado: estabelecimento de acordos de confiança entre várias instituições que confiam entre si para integrarem os seus IdP numa rede de confiança mútua — os IdP federados emitem alegações de identidade de forma integrada e compatível

Descentralizado: os pares estabelecem uma conexão, que lhes permite comunicarem livremente entre si sem necessitarem de recorrer a intermediários externos, através de um canal de comunicação seguro, associado ao par, que não é partilhado por mais ninguém e que persiste enquanto ambos assim o entenderem DID

PROTOCOLOS DE IAA

SAML: permite a autenticação e identificação de entidades que ~~guardam~~ a um SP/RP e cuja identidade é gerida por um IdP em regime de SSO

OAuth 2.0: protocolo de autorização desenvolvido para aplicações web que permite a aplicações terceiras acederem a recursos específicos de um utilizador numa outra aplicação ou plataforma, sem necessidade de conhecer a senha desse utilizador

OpenID Connect (OIDC): estende o OAuth 2.0 para permitir que os RP obtenham informações básicas do perfil do utilizador de forma normalizada — autenticação

OpenID para Credenciais Verificáveis (OID4VC): estende o OIDC para permitir que as VC sejam emitidas e apresentadas como parte de um fluxo de autenticação

SELF-SOVEREIGN IDENTITY

Self-Sovereign Identity (SSI): os utilizadores devem ser capazes de poder criar, controlar e gerir a sua própria identidade, de forma o mais autónoma/soberana e descentralizada possível, sem que para isso dependam de nenhuma autoridade sem ser a sua própria — é possível autenticar qualquer entidade participante, auto-declarar alegações, receber e gerir alegações de terceiros e partilhá-las sem dependências de terceiros

1. **Existência:** as entidades têm uma existência própria independente
2. **Controle:** as entidades controlam as suas identidades
3. **Acesso:** as entidades têm acesso total aos seus próprios dados
4. **Transparência:** os sistemas/algoritmos usados são conhecidos e auditáveis
5. **Persistência:** as identidades são de longa duração
6. **Portabilidade:** a informação e os serviços são transportáveis e seguem o titular
7. **Interoperabilidade:** a identidade pode ser usada de forma generalizada
8. **Consentimento:** as entidades têm de concordar com o uso da sua identidade
9. **Minimalização:** a obrigatoriedade/necessidade de divulgar "claims" deve ser minimizada
10. **Proteção:** os direitos legais das entidades são protegidos

Identificador Descentralizado (DID): identificador normalizado que permite a qualquer entidade tirar partido de uma identidade digital verificável de forma totalmente descentralizada

1. **Permanência:** um DID é imutável e único
2. **Resolvível:** um DID é um URN resolvido na Internet para obter os seus metadados
3. **Verificável:** o controlador/dono consegue provar de forma criptográfica o controlo
4. **Descentralizado:** não é necessário recorrer a uma terceira parte confiável
5. Qualquer entidade pode ter quantos DIDs quiser e usá-los convenientemente

- Um DID pode estar relacionado a qualquer entidade em qualquer contexto

Controlador/Dono: entidade que controla a chave privada associada ao DID
Metadados: chave pública e endpoints de serviço do DID (URLs)

Documento DID: contém uma ou mais chaves públicas para autenticação do sujeito, um ou mais serviços associados ao sujeito para interações e outros metadados

Método DID: define a sintaxe e as operações Create Read Update Delete (CRUD)

☹ PKI: custo; atrito; ponto único de falha; alteração de identificadores e chaves públicas
☺ DID: descoberta segura de endpoints; conexões seguras DID2DID; privacidade

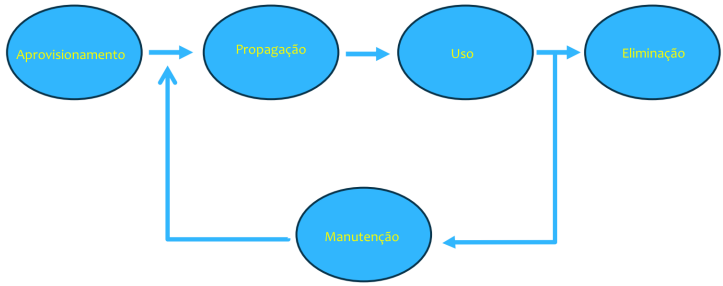
Credencial Verificável (VC): documento estruturado que representa credenciais inalteráveis cuja autoria pode ser validada por meios criptográficos — forma padronizada de representar "claims" que um emissor faz sobre o titular da credencial

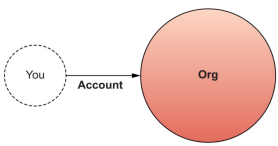
Verificação: pelos verificadores, no formato de Apresentações Verificáveis
Autenticidade: através de meios criptográficos e de validação de assinaturas

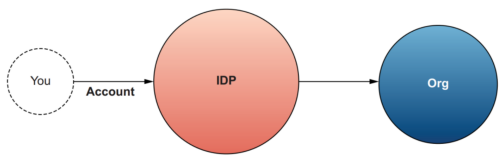
- O uso de VC implica a existência de um registo de dados verificáveis que atua como mediador na criação, verificação criptográfica descentralizada e resolução dos identificadores — registo de revogações, lista dos DID dos fornecedores, ...
- Este modelo permite custódia digital
- A informação a apresentar para um verificador ou RP deve ser a mínima
- As VCs podem incorporar diferentes esquemas criptográficos para validação/prova

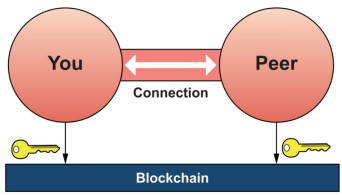
Carteira Digital: software que permite ao controlador da carteira gerar, armazenar, gerir e proteger chaves criptográficas, segredos e outros dados privados sensíveis

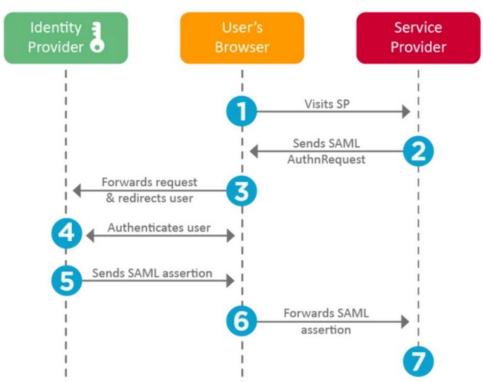
Agente Digital: software que permite a um controlador realizar ações, comunicar e trocar dados, armazenar informações e acompanhar o uso da carteira digital

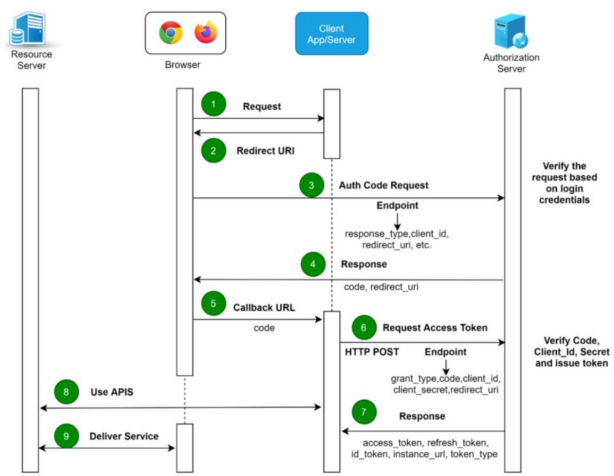

Aprovisionamento
Propagação
Uso
Eliminação
Manutenção

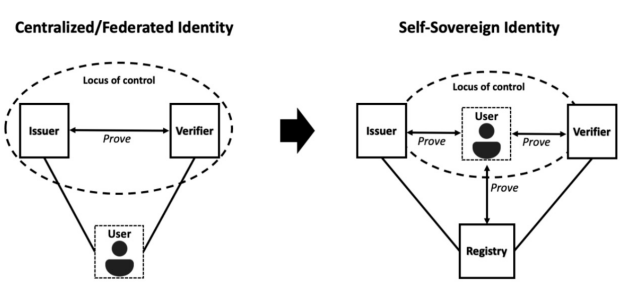

You
Account
Org

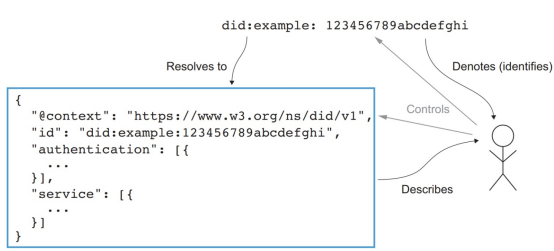

You
Account
IDP
Org


You
Peer
Connection
Blockchain


Identity Provider
User's Browser
Service Provider
1
Visits SP
2
Sends SAML AuthnRequest
3
Forwards request & redirects user
4
Authenticates user
5
Sends SAML assertion
6
Forwards SAML assertion
7


Resource Server
Browser
Client App/Server
Authorization Server
1 Request
2 Redirect URI
3 Auth Code Request
Endpoint
response_type,client_id, redirect_uri, etc.
Verify the request based on login credentials
4 Response
code, redirect_uri
5 Callback URL
code
6 Request Access Token
HTTP POST Endpoint
grant_type,code,client_id, client_secret,redirect_uri
Verify Code, Client_Id, Secret and issue token
7 Response
access_token, refresh_token, id_token, instance_url, token_type
8 Use APIS
9 Deliver Service


Centralized/Federated Identity
Locus of control
Issuer
Prove
Verifier
User
Self-Sovereign Identity
Locus of control
Issuer
Prove
User
Prove
Verifier
Prove
Registry


did:example: 123456789abcdefghi
Resolves to
Denotes (identifies)
{
"@context": "https://www.w3.org/ns/did/v1",
"id": "did:example:123456789abcdefghi",
"authentication": [{
...
}],
"service": [{
...
}]
}
Controls
Describes

1. **(1)** O que entende por "linha de água alta" e "linha de água baixa" no contexto de um MLS?

2. **(2)** Explique o que é o Chord, e as suas diferenças para o kademlia explicando os processos internos do Chord.

3. **(2)** No contexto de 2AF, descreva 4 dos possíveis ataques ao SS7, e globalmente qual o impacto para o seu uso em 2AF.

4. **(2)** Explique as vantagens de usar FIDO num sistema Federado de autenticação, mostrando o diagrama para um cenário típico de utilização, explicando todas a entidades envolvidas.

5. **(2)** Descreva o problema descrito no paper "The Confused Deputy", dando de seguida uma solução para a sua mitigação.

6. **(2)** Descreva o modelo de k-anonymity, explicando o que é um quasi-identifier, e por fim 3 ataques contra o modelo.

(2) Explique qual o propósito do Stuxnet, enumerando os vectores de ataque utlizados e o mecanismo físico onde por fim actuou.

(2) Explique o que é um rootkit, listando as 3 categorias de classificação, dando um exemplo (com descrição) por categoria.

) Diga se um programa pode ser completamente ofuscado, justificando a sua resposta.

Explique o que é um sistema DRM, detalhando a sua relação com criptografia e "scrambling" fim um exemplo da sua implementação.

EXAME

1. Um modelo de Segurança Multi-Nível (MLS) é uma forma de controlo de acesso que é necessária quando sujeitos e objetos com/em níveis diferentes de "clearance" e classificação, respetivamente, utilizam o mesmo sistema, coexistindo. Estes modelos — como é o caso do Bell-LaPadula — explicam o que tem de ser feito, mas não como implementar, pelo que são descritivos e não prescritivos.

Os princípios de "linha de água alta" e "linha de água baixa" aplicam-se aos modelos/sistemas MLS nos quais se admite que a etiqueta de segurança (para sujeitos e objetos) seja alterada conforme necessário, mas se e só se essa alteração não violar a política de segurança estabelecida, de acordo com a propriedade de tranquilidade fraca.

Em particular, o princípio de "linha de água alta" significa que, de acordo com o princípio de privilégio mínimo, deve ser atribuído aos utilizadores o menor privilégio — neste caso, "clearance" — possível para a realização do seu trabalho e o desempenho das suas funções, atualizando-o conforme necessário e permitido pela política. Assim, a "linha de água alta" corresponde ao nível mais elevado de "clearance" que um sujeito pode obter, de acordo com as suas necessidades e com o estabelecido pela política, de maneira a respeitar o princípio de privilégio mínimo. Em sentido inverso, a "linha de água baixa" corresponde ao menor nível de "clearance" atribuível a um sujeito, analogamente.

Note-se que, no MLS Bell-LaPadula, os utilizadores só podem ler objetos com classificação inferior ou igual à sua "clearance" e só podem escrever objetos com classificação superior ou igual à sua "clearance", pelo que estes princípios de linha de água alta e baixa permitem respeitar o princípio de privilégio mínimo, admitindo a propriedade de tranquilidade fraca.

2. O Chord é um protocolo de Distributed Hash Table (DHT) usado em sistemas e redes Peer-to-Peer (P2P) para localizar nós e dados de forma eficiente numa rede distribuída e escalável. Assim, o objetivo principal do Chord para permitir a localização eficiente do nó responsável por uma determinada chave num anel lógico de nós.
Internamente, o Chord considera que cada nó e/ou chave é mapeado para um identificador único através de uma função de hash. Deste modo, os identificadores são dispostos num anel circular em que cada chave é atribuída ao primeiro nó cujo ID é superior ou igual ao ID da chave. Além disto, cada nó armazena uma "finger table" que, numa operação de pesquisa, permite encaminhar o pedido para saltar progressivamente mais longe dentro do anel. Finalmente, o protocolo estabelece que as "finger tables" devem ser atualizados sempre que um nó entra ou sai da rede, executando estabilização periódica para garantir consistência.
O Kademlia é outro protocolo DHT para redes/sistemas P2P, com diferenças relativamente ao Chord. Em primeiro lugar, enquanto o Chord considera um anel circular para o cálculo das distâncias entre os nós, o Kademlia considera como métrica de distância o resultado da operação XOR entre os IDs dos nós. Em segundo lugar, enquanto cada nó Chord armazena uma "finger table" para roteamento, no Kademlia cada nó contém a sua "routing table" com um dado número de K-buckets, que armazenam nós. Em terceiro lugar, a pesquisa no Kademlia é efetuada em paralelo, ao contrário do Chord, em que é sequencial. Por último, no Chord a rede é estabilizada periodicamente, mas, no Kademlia, isto é feito através de atualizações passivas de acordo com as interações entre nós. Fundamentalmente, destaque-se que o Kademlia não assume/considera uma rede num anel circular, como faz o Chord. Na prática, o Chord é mais simples e estruturado, mas o Kademlia é mais eficiente e resiliente.

3. O Signaling System 7 (SS7) é um protocolo de comunicação entre redes e operadoras de telecomunicações, utilizado para o envio e receção de mensagens SMS. No contexto de Autenticação de Dois Fatores (2AF), o SMS é tradicionalmente usado como segundo fator de autenticação - complementar à palavra-passe -, apesar de não ser tão seguro como aparenta, tal como demonstram os seguintes quatro exemplos de ataque ao SS7:

(1) "Interception": o atacante é capaz de intercetar SMS para utilizadores legítimos, potencialmente com códigos de 2AF, ao enviar pedidos de redirecionamento/encaminhamento de mensagens SMS através da rede SS7;

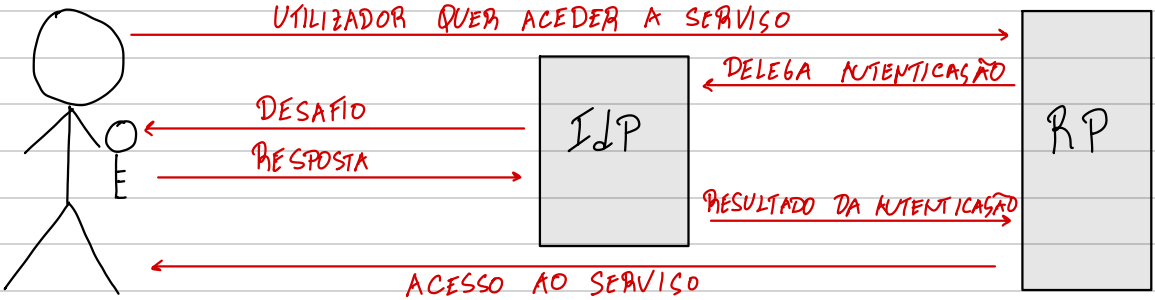
(2) "Tracking": o atacante é capaz de rastrear a localização dos dispositivos dos utilizadores usado para a receção de códigos 2AF, ao enviar mensagens SMS com pedidos "Provide Subscriber Information" (PSI);

(3) "Wiretapping": o atacante é capaz de escutar chamadas telefónicas que podem conter códigos 2AF para autenticação ao monitorizar os bits trocados na rede, a partir dos quais é possível extrair informação;

(4) "Spoofing": o atacante é capaz de falsificar a sua identidade para assumir-se outro utilizador ou uma entidade legítima, de maneira a enviar ou receber códigos que servem como 2AF, indevidamente;

O impacto destes ataques em 2AF é crítico, porque inviabiliza a utilização segura de SMS e chamas telefónicas para o envio e receção de códigos de autenticação, dado que podem estar a ser intercetadas, rastreadas, escutadas ou falsificadas por um atacante. Assim, a autenticação complementar por estes meios deixa de ser confiável, o que leva à necessidade de implementar, adotar e seguir outros meios de autenticação modernos, como aplicações móveis e/ou chaves físicas.

4. O FIDO é um conjunto de especificações que define um modelo/sistema/fluxo moderno de autenticação, particularmente usado e vantajoso em contexto federado.
Num sistema federado de autenticação, a autenticação é delegada de uma Parte Confiável (RP) para um Fornecedor de Identidade (IdP) que é responsável por autenticar o utilizador e transmitir toda a sua informação necessária às RPs.
O FIDO é utilizado para autenticar o utilizador no IdP.



(1) Utilizador: cliente que pretende aceder a um serviço e possui Autenticador
(2) Autenticador: dispositivo certificado pela FIDO, suportado/confiado pelo IdP e aceite/permitido pela RP para responder a desafios de autenticação do Utilizador
(3) IdP: Fornecedor de Identidade, que autentica o Utilizador pelo Autenticador
(4) RP: Parte Confiável, que disponibiliza o serviço e a política para aceder

Note-se que, num sistema federado, o mesmo IdP pode ser utilizado para autenticar o utilizador perante múltiplas RPs diferentes.

As vantagens da utilização de autenticação FIDO neste sistema são: (1) eliminação das palavras-passe e substituição por meios de autenticação mais fortes e robustos (como biometria e/ou tokens físicos), o que garante (2) resistência a phishing, (3) conveniência para o utilizador, (4) interoperabilidade e compatibilidade entre diferentes plataformas, dispositivos e serviços e (5) privacidade do utilizador, cuja autenticação é feita através do autenticador.

5. O problema "The Confused Deputy" ocorre quando, num sistema de controlo de acesso gerido por Listas de Controlo de Acesso (ACLs), um sujeito é capaz de colocar outro sujeito a agir em seu nome perante determinados objetos, de modo que este fique confuso relativamente aos direitos e permissões a exercer sobre os objetos, isto é, confunda as suas próprias capacidades com as do sujeito original/inicial. Por exemplo, se a Alice não pode escrever (n)um ficheiro, mas pode executar o Compilador e o Compilador, por sua vez, pode escrever (n)esse ficheiro, então a Alice invocar o Compilador para escrever (n)esse ficheiro, sendo este um caso clássico do problema "The Confused Deputy", no qual os direitos do Compilador se confundem com os da Alice. Assim, existe uma separação da autoridade do propósito para o qual ela é usada.
Uma solução para a mitigação deste problema para por implementar um sistema de controlo de acesso baseado em Capacidade, em vez de ACLs. Num sistema por Capacidades, em vez de o próprio objeto conter uma lista de sujeitos permitidos (ACL), cada sujeito é que detém uma lista de objetos para os quais tem permissões. Assim, para aceder a um recurso/objeto, o próprio utilizador/sujeito tem de fornecer e demonstrar as credenciais necessária para o efeito, pelo que se torna fácil delegar a autoridade, bastando para isso, no exemplo dado, que a Alice passe ao Compilador as suas Capacidades para executar em seu nome, o que resultaria na impossibilidade de escrever (n)o ficheiro, sem qualquer confusão. Assim, evita-se o problema "The Confused Deputy".

6. O modelo de "K-anonymity" é um modelo para a publicação de dados com preservação da privacidade, que visa impedir que os indivíduos possam vir a ser reidentificados a partir da publicação de conjuntos de dados anonimizados. Assim, este modelo define/estabelece que um registo X é considera "K-anónimo" se for indistinguível de, pelo menos, K-1 outros registos do conjunto de dados. Assim, um atacante não consegue reidentificar um indivíduo dentro de um conjunto de K registos.

Neste contexto, um quase-identificador (QID) é um atributo que, não identificando diretamente um indivíduo (como um nome ou número de identificação) é capaz de, quando combinado com outros atributos e/ou com dados públicos, auxiliar na reidentificação de um indivíduo com alta probabilidade. Por exemplo, a data de nascimento é um QID típico.

Existem três ataques típicos contra o modelo "K-anonymity":

(1) Ataque de Homogeneidade: quando todos os registos de indivíduos pertencentes à mesma classe de equivalência (K indistinguíveis) têm o mesmo valor do atributo sensível, pelo que o atacante o conhece/obtém imediatamente;

(2) Ataque de Conhecimento de Fundo: quando o atacante usa conhecimento externo previamente adquirido para distinguir entre registos aparentemente indistinguíveis numa classe de equivalência, contribuindo para facilitar a reidentificação;

(3) Ataque de Ligação: quando o atacante cruza informação do conjunto de dados anonimizado com outros conjuntos de dados públicos, potencialmente com os mesmos QID, facilitando a reidentificação

7. O Stuxnet é um worm (malware - software malicioso) cujo propósito para por sabotar fisicamente as instalações nucleares, em particular as máquinas industriais (OT) Siemens associadas ao enriquecimento de urânio, como sistemas Scada e PLC.

O ataque elaborado para o Stuxnet utiliza uma cadeia de vetores de ataque, nomeadamente: (1) infeção através da introdução de "pen" USB em dispositivos "offline", (2) exploração de vulnerabilidades "zero-day" no Windows para a execução remota de ficheiros, o escalamento de privilégios e a permanência, (3) propagação nas redes internas industriais das organizações através de serviços do Windows e de técnicas como pass-the-hash para roubar credenciais, (4) a utilização de certificados digitalmente assinados para dissimulação e (5) a interação com os sistemas OT para o envio de comandos modificados e maliciosos.

Após toda esta exploração, o mecanismo físico no qual o Stuxnet atuou foi nas máquinas centrifugadoras das indústrias de produção de energia nuclear, em particular nos sistemas OT Scada e PLC da Siemens.

8. Um rootkit é um tipo de malware (software malicioso), que consiste num conjunto de programas para obter acesso com permissões/privilégios de administrador a um dispositivo. Este tipo de malware é capaz de alterar o sistema operativo em que se encontra instalado de modo a subverter o seu funcionamento, assim como esconder a sua existência para evitar ser detetado.
Um rootkit pode ser classificado numa de três categorias:
(1) User-Mode: o rootkit atua ao nível da aplicação, modificando as bibliotecas do sistema ao interetar chamadas às respetivas funções e adulterá-las; por exemplo, o rootkit Hacker Defender existe ao nível do utilizador e é capaz de esconder processos, ficheiros e registos, sendo controlável remotamente;
(2) Kernel-Mode: o rootkit atua ao nível da/do Kernel do sistema operativo, interetando as chamadas ao sistema e potencialmente controlando os drivers; por exemplo, o rootkit Adore-Ng injeta-se na/no Kernel Linux através de módulos e é capaz de ser usado em ataques avançados por ameaças persistentes (APT);
(3) Firmware: o rootkit atua abaixo do sistema operativo, no próprio firmware do dispositivo, pelo que se torna persistente entre formatações, logo, extremamente difícil de detetar e remover; por exemplo, o LoJax compromete a UEFI de maneira a penetrar de forma profunda e persistente nos sistemas, também associado a APT.

9. A ofuscação de código consiste em tornar o código-fonte de um programa (mais) difícil de perceber/compreender, tradicionalmente para evitar ataques de engenharia reversa. No entanto, não é possível ofuscar completamente um programa! A prova para esta afirmação existe num artigo previamente publicado, resumindo-se ao facto que não é possível existir um "ofuscador" que, simultaneamente, preserve toda a funcionalidade do programa a ofuscar, mas que o faça comportar-se como uma "caixa negra" a partir da qual não é possível extrair qualquer informação. Efetivamente, submeter o código a alterações, mas garantindo a preservação da sua funcionalidade, resultará sempre em código funcional que, com maior ou menor esforço por parte do atacante/adversário, pode sempre ser submetido a um processo de engenharia reversa para compreender o seu funcionamento interno original. Assim, por mais que se alterem os nomes das variáveis e funções, se ofusquem os predicados das expressões, se insira código morto, entre outras técnicas, o código binário em execução, desde que funcional, poderá sempre ser descontruído por um atacante para o compreender. Por isso, a ofuscação de código não é uma solução de segurança inviolável, mas apenas auxilia, ao atrasar potenciais atacantes, dificultando-lhes o trabalho.

10. Um sistema Digital Rights Management (DRM) é uma tentativa de aplicar um conjunto de políticas e de tecnologias para distribuir conteúdo digital, mas manter algum controlo remoto no seu uso, após a entrega. Assim, DRM visa resolver o problema da proteção permanente à informação digital, ou seja, forçar restrições ao uso do conteúdo (como limitações de cópia, leitura, reencaminhamento, entre outros), após a entrega.

Para cumprir o seu propósito, um sistema DRM recorre a criptografia e a "scrambling". Por um lado, a criptografia, embora não seja, por si só, a solução para este problema, é necessária para entregar o conteúdo de forma segura e para prevenir ataques triviais. Assim, a criptografia permite encriptar o documento em trânsito para o proteger contra atacantes que o intersetem, garantindo a sua integridade e confidencialidade na entrega, mas não resolve o problema na totalidade. Por isso, surge o "scrambling" que, utilizando algoritmos semelhantes aos criptográficos, permite baralhar os sinais digitais de modo a dificultar a sua partilha pelo próprio utilizador detentor da chave de desencriptação, exceto por meios analógicos de cópia. Assim, o "scrambling" torna os dados inutilizáveis sem a chave correta, ao garantir que o "scrambling" é realizado no servidor e o "de-scrambling" no cliente, mas por um algoritmo confiável e seguro, provavelmente proprietário, que ofusca os dados de modo a dificultar a sua reprodução indevida.

Por exemplo, plataformas de alojamento, partilha e reprodução de conteúdo como o Netflix recorrem a algoritmos criptográficos e de "scrambling" para implementar um sistema DRM, garantindo que os seus filmes e séries só são visualizados por utilizadores autorizados. Para isso, o utilizador deve autenticar-se perante a Netflix para receber uma chave de desencriptação e "descrambling" do conteúdo, que se mantém sempre encriptado e baralhado até à sua reprodução no dispositivo. A impossibilidade de cópia indevida é forçada através de restrições aplicadas ao nível do software e do hardware, pelo próprio sistema DRM.

Computer Networks 94 (2016) 205–218

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

S-Kademlia: A trust and reputation method to mitigate a Sybil attack in Kademlia

Riccardo Pecori*

University of Parma, Department of Information Engineering, V.le G.P. Usberti 181/A, 43124 Parma PR, Italy

ARTICLE INFO

Article history:
Received 11 January 2015
Revised 14 October 2015
Accepted 6 November 2015
Available online 12 November 2015

Keywords:
Peer-to-peer
Sybil attack
Incorrect routing
Kademlia
Trust and reputation
Storage and retrieval attacks

ABSTRACT

Peer-to-peer architectures have become very popular in the last years for a variety of services and applications such as collaborative computing, streaming and VoIP applications. The security and integrity of the overlay involved in such networks is a fundamental prerequisite for deploying such a technology. Withstanding multiple false identities in the overlay, also known as a Sybil attack, is one of the main challenges in securing structured peer-to-peer networks. Poisoning routing tables through these identities may make the routing and storage and retrieval processes extremely difficult and time consuming. In this paper we investigate possible countermeasures and propose a novel method for making the routing and the storage and retrieval of resources in a Kademlia network more secure through the use of a combined trust-based algorithm exploiting reputation techniques. Our solution provides a balanced mixing of standard Kademlia algorithms and trust-based algorithms showing promising results in thwarting a Sybil attack in a Kademlia network, in comparison with similar methods as well.

1. Introduction

In the current structure of the Internet, peer-to-peer (P2P) networks have conquered a significant part in the whole traffic distribution and they have become very popular for their scalability and the great amount of services they can support. Upon their inception, they were mainly deployed as a simple, decentralized and scalable way to exchange files, but they are now used for several different services as well. Among these we can recall P2P Voice-over-IP (VoIP) communications [1,2], streaming applications, sharing bandwidth and computing power, storage capacity and many others.

During the last years, in particular, the role of structured P2P networks has been becoming preponderant; this is mainly due to their ability of both organizing the network, through their own addressing schemes, into a so called "overlay network" and of exploiting particular algorithms for fast and efficient lookup and storage and retrieval operations. Some examples of these algorithms are Chord [3], Kademlia [4], Pastry [5], CAN [6], and the like. They use Distributed Hash Tables (DHTs) to allow for efficient lookup of identifiers and routing to the corresponding nodes. This is achieved by imposing a strict structure on the routing tables of nodes, warranting quick convergence to a target. This firm framework makes DHTs, on one side, efficient and simple to use, on the other side, subject to be easily attacked and broken by a set of malicious nodes that return not useful information instead of helping in the routing. As a matter of fact, a very large number of fake identities (IDs) in the overlay can poison honest nodes routing tables and disrupt or degrade the DHT performances. This is the so-called "Sybil attack" [7].

Therefore some security issues must still be fixed, and this is the purpose of this paper that takes place in this scenario where we studied possible solutions for avoiding or better mitigating the misbehavior caused by sybil nodes. Particularly we considered an application context in which the P2P routing scheme is based on the classic routing structure of

* Tel.: +39 3920989482.
E-mail address: riccardo.pecori@unipr.it, riccardo.pecori@gmail.com

http://dx.doi.org/10.1016/j.comnet.2015.11.010
1389-1286/

Kademlia [4], one of the most widespread DHTs (so much that it is used, for example, by eMule, BitTorrent, etc.).

Moreover, we did not try to prevent the creation of sybil nodes, rather, we supposed the environment to analyze is inherently infected by sybils, i.e., there is a subset of nodes that can present multiple identities. So, like in [8], we are not interested in a "clean network" but in a "trusted network", in which only the most trustworthy nodes should be used for both routing and storage and retrieval operations. A node routing table may and even needs to contain nodes it does not trust. The requesting node shall be able to decide on its own which nodes to trust in order to store its data or to retrieve data from, on the basis of a quality rating rooted in a trust score. Moreover, we allowed the decision, about which nodes are trustworthy and which are not, to vary between nodes, so we considered a local trust.

So, following this more conservative and robust approach, we re-adapted Kademlia standard procedures (both routing and storing) in order to take into account trust and reputation as well. This allows the reordering of the temporary search list and the final storing list in such a way that the sybils, if they are a limited percentage of all nodes, cannot completely degrade performances. Our solution is based on security measures that have proven themselves effective in other contexts and we combined these measures in a new way, applying them to a structured Kademlia network and making opportune adaptations. With our proposal we got a more reliable and secure Kademlia network, called S-Kademlia, which, even if not the optimum, makes the routing and storing mechanisms stronger against the presence of sybil nodes in the network.

To summarize, the main contributions of our work concern:

1. the evaluation, through a detailed survey, of both current and past anti-Sybil solutions,
2. the assessment of the degradation of Kademlia standard algorithm in presence of a growing number of sybil nodes with different malicious behaviors,
3. the proposal and evaluation of a new algorithm mixing the standard Kademlia routing and storage/retrieval procedures with a proper trust scheme, and
4. the comparison of this new algorithm with one already presented in the literature and based only on trust.

The rest of the paper is organized as follows: in Section 2 the background about the Sybil attack and possible countermeasures is described, in Section 3 there is a brief summary of the classic Kademlia routing and storage and retrieval procedures along with an analysis of sybils effects on them, in Section 4 we accurately present our integrated trust-based algorithm besides the trust management, Section 5 presents in details simulation results, whereas Section 6, in the end, seals up the work with some conclusions.

2. Sybil attack: rationale and countermeasures

Peer-to-peer networks usually rely on the existence of multiple, independent, and remote entities to mitigate the threat of bogus peers. The Sybil attack [7] takes advantage

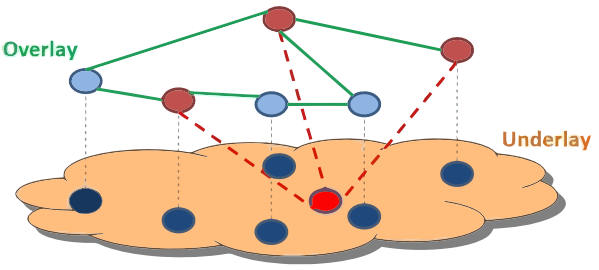


Fig. 1. Sybil attack.

of this feature, commonly present into P2P systems, as it is normally started by a faulty entity masquerading with multiple identities in the overlay network. The key idea behind the Sybil attack is to insert in the overlay malicious identities, the *sybils*, which are all controlled by one single physical entity (Fig. 1). This allows, provided that the sybils are positioned into strategic places, to gain control over a part of the P2P network or even over the whole network, to monitor the traffic or to abuse of the DHT protocol in other ways. This can lead to other types of attacks such as the eclipse attack [9]. As already stated in the previous section, we assume our scenario is inherently affected by sybils, so threats coming from a malicious behavior of these nodes are very probable.

The threats they can generate may be roughly divided into the following three categories [10]:

- routing attacks,
- storage and retrieval attacks,
- miscellaneous attacks.

The first group of attacks can take various forms such as incorrect lookup routing, incorrect routing updates for the routing tables and partition into an incorrect network. The second type of attacks can be represented by the following behaviors: denying actually stored data or pretending to have resources not owned, for example by providing fake resource data. The third kind is made up of different malicious attitudes that cannot be classified in the previous two points: inconsistent behavior, overload of target nodes, churning or sending of unsolicited messages are examples of these kinds of threats. In our work we only focused on incorrect routing attacks, partly studied in [11], and storage and retrieval attacks, leaving other attacks out of the scope of this paper for future work.

Currently no one has found a general and effective distributed solution to the Sybil attack despite many general approaches presented in the literature. Our purpose was, after analyzing the effectiveness of various proposed solutions, to devise an effective scheme applicable to a particular structured P2P network, Kademlia. So, first of all, we present in the following subsection a complete analysis of current known countermeasures to different misbehaviors that can be caused by a Sybil attack and then, in the following sections, we highlight a possible complete and effective countermeasure for Kademlia based upon trust and reputation.

2.1. Countermeasures to the Sybil attack

The current solutions to a Sybil attack, which can be found in the literature, can be roughly classified, according to [12], into four categories:

- trusted certification,
- resource testing,
- costs and fees,
- trusted devices.

This is a classification for solutions to avoid sybils inside a network, but as already stated in Section 1, we hereby consider a different point of view, therefore sybils are present in our network scenario. Considering the specific threatened DHT operations, anti-sybils solutions may also follow an alternative classification, such as the following:

1. usage of diverse routing schemes (against routing attacks),
2. limitation on the number of peers that a physical node can admit to the network (against join attacks),
3. verification of some types of requirements, such as computational resources (against join and store attacks),
4. periodical refresh of the routing tables or of the IDs (against routing and storage/retrieval attacks).

The presence of a centralized Certification Authority (CA) is also claimed to be one of the best solutions, if not the only one [7]. The whole analysis of countermeasures as well as their advantages and drawbacks is investigated in Table 1. The study is done according to different levels: considering the specific threatened DHT operations (mainly *find_node*/*find_value* and *join* procedures), then the malicious activities involved and then the general approach of the solution. Finally a brief description of the solution along with some advantages and drawbacks is given.

In the following a brief summary of the findings highlighted in Table 1 is provided.

As concerns the DHT *find_node*/*find_value* procedure, it may result vulnerable to incorrect routing attacks or incorrect routing update attacks. The possible solutions [13] regard the usage of redundancy techniques, where one can find diverse routing strategies, not based upon the classic closeness routing, or techniques that exploit further constraints, such as the number of intersections of random routes, failure tests on the routing primitive, etc.

Regarding incorrect routing updates, instead, the possible solutions [14] are based upon a general resource testing scheme, upon redundancy or upon trusted certification. For the first countermeasure identity legitimacy controls can be enacted through messages sent in pre-assigned channels or through connectivity degree of a node. For the redundancy solutions more than one routing table can be used and finally for the trusted certification the possession of some cryptographic material. A specific malicious activity against the *find_value* DHT procedure is the storage and retrieval attack. Against this type of attack one possible solution can be the testing of resources, for example through storage or computation operations [7].

As regards the *join procedure*, it is usually attacked through the creation of false identities in the overlay network. The possible countermeasures are limitation-based, trust-certification-based, resource-testing-based and cost-and-fee-based. Concerning the first ones, examples can be found in limiting the join permission for active peers, for the second one in the presence of central authorities issuing identities, maybe with the help of certificates or through different channels. Regarding the second type of countermeasures also the periodical invalidation of IDs by a trusted entity or the registration of a new ID by other nodes may be considered. For the third type of countermeasures, resource testing, there can be mechanisms that provide an admission control, hierarchical or distributed, with puzzle challenges, or that bind the virtual ID to real world identities. Finally "costs and fees" countermeasures are based upon certificates that must be paid with real money.

After this analysis we propose a distributed solution scheme, in order to maintain the inherently non-centralized nature of the P2P networks, based upon trust and reputation according to some successful suggestions found in [22]. Refs. [23,24] are frequently called upon as the best schemes, while [25] limits the application of trust to P2P file sharing in unstructured networks. The solution in [26] operates in structured networks but does not have an implementation in the real world. In [8] the author presents a first approach of using a trust metric within the Kademlia routing process, whereas in [27] he extends his ideas also to the store/retrieval process, making, however, a distinction between trust for LOOKUPs and for GET and PUT operations. Another work very close to ours is [28] where, nevertheless, certificates and private and public keys are used. Our idea is simpler in the formulas computing the trust and does not need the usage of any thresholds or Certification Authorities. The ideas presented in [27], moreover, introduce, as stated also by the author, a lot of overhead and still lack of simulation results and analysis. Anyway, the first papers, at the best of our knowledge, trying to use trust and reputation for solving routing and storage attacks caused by sybils in Kademlia are given in [8,27], and for this reason we will compare our algorithm to the ones presented in these works. Our balanced trust-based scheme is adapted to the classic Kademlia algorithm (both for routing and for storage and retrieval) and, more important, it is a completely P2P solution, in order to maintain the inherently not centralized nature of Kademlia. As regards trust and reputation we took advantage of some successful suggestions found in [22,29–31]. In this paper we also compare our balanced trust-based algorithm with the one in [27] and show that it reaches better performances in terms of successful operations in a similar risk environment.

3. Consequences of a Sybil attack in Kademlia

In this section we deeply analyze the effects sybils cause in a Kademlia network. In the following we first briefly recall Kademlia classic routing and storing procedure [4], then, by means of our simulation results, we analyze how Kademlia performances degrade in a Sybil scenario, showing the effects of these malicious nodes on the standard Kademlia algorithm.

Table 1
Analysis of the main methods to defend networks against sybil behaviors.

DHT opers.	Mal. activity	Solut. type	Protection technique	Advantages	Drawbacks	Ref.
			Diversity routing: ID lookup trust profile (histogram representing nodes appearance frequency on the lookup path towards the ID)	Adversary nodes introducing a lot of sybils may acquire high values in the trust profile, so nodes behind them will not be used	Inefficient under normal circumstances and no progress toward target node	[13]
			Mixed routing: a balance between classic routing and diversity routing	Balance between standard and diversity approach	It does not perform better than the diversity strategy in case there are many more sybils than honest nodes	[13]
Find node/find value	Incorrect routing	Redundancy-based	Zig-zag routing: alternation of diversity and classic routing	Outperforms standard and mixed routing when the number of malicious nodes is large	Reduced lookup speed	[13]
			Secure routing primitive: failure test and redundant routing to ensure, with high probability, that at least one message arrives at each correct replica root for a key	Solves a constrained routing problem: even a few malicious nodes can reduce the probability of successful delivery	Several attacks have been proven to weaken the failure test	[14]
			Random routes with a predefined number of hops; identity verification based on two random routes; check of number of intersections	No centralization elements; trust and reputation policies are applicable due to the character of nodes relationships (social network)	Routing loops reduce the effective length of the routes and the probability of intersections; small number of attack edges assumed; honest users may be compromised; the number of hops as critical parameter	[15]
			Use of 2 routing tables: one exploiting network proximity, another exploiting a constraint on closeness	Constrained routing tables have an average fraction of only a few random entries pointing to attacker-controlled nodes	Routing overhead, an attacker can reduce the successful delivery probability not forwarding messages; no flexibility in neighbor selection	[14]
		Redundancy-based	Periodic reset of optimized routing tables to constraint tables; tables updates rate limitation; unpredictable ID assignment.	Adversaries must make more intensive their activity to maintain their foothold.	Short-term effect; only Bamboo tested; effective at low routing tables poisoning level; system adaptability and routing security trade-off ; randomness server availability.	[16]
Find node/find value	Incorrect routing update		Identity legitimacy by sending check messages at time intervals on channels pre-assigned to neighbor nodes.	No centralization elements involved.	A node cannot communicate simultaneously on more than one channel.	[17]
		Resource testing	Node degree bounded to a threshold: nodes anonymously audit the connectivity of each other through certificates binding ID and public key.	Flexibility in the selection of neighbors nodes.	Mainly addressed to eclipse attack; mechanisms for maintaining auditor anonymity required; localized eclipse attacks could work.	[9]
		Trusted certif.s/devices	Random assignment of key-sets to each node; sybils detection checking the number of coinciding keys in sets of different nodes.	Distributed control mechanisms.	Possibility to stole IDs (masquerading attacks).	[17]
Find value	Storage and retrieval attack.	Resource testing.	Resource control through simultaneous check of minimum resource level: communication, storage and computation operations.	No centralized elements.	It is not practically feasible in large-scale distributed systems.	[7]

(*continued on next page*)

Table 1 (*continued*)

DHT opers.	Mal. activity	Solut. type	Protection technique	Advantages	Drawbacks	Ref.
			Hierarchy of cooperative admission controls based on the bootstrap graph. Challenging new nodes by puzzles solving in order to have a global cryptographic proof. Upper bound on the number of IDs.	Distributed access control; more difficult targeted attacks; easy malicious node removal by pruning; easy attack localization.	Simple Diffie–Hellman used to share a secret in the hierarchy: possible man in the middle. Computationally expensive signature verification. Tree root as single point of failure.	[18]
Join	False ID creation	Resource test	Requiring the prospective (joining) node to solve crypto puzzles.	Distributed access control.	Costs trade-off in the solving of puzzles: not too expensive for good nodes but enough to prevent attacks.	[14]
			Binding node IDs to real-world identities.	Effective in "virtual private" overlays (enterprise networks).	CA is a single point of failure.	[14]
			Random computational puzzles, with incorporated distributed locally-generated challenges, required to participate in the network; challenge broadcasting from each peer toward others.	Locally challenges prevent puzzles to be reused by attackers over time; gives a node flexibility to choose its position.	Opposite effect when legitimate users are stuck with out-of-date hardware whereas attackers can rely on high performance computers.	[19]
		Constraint	Limited join permission by active P2P network nodes.	A limited possibility for introducing sybils into the network through the same bootstrap-node.	Introduction of additional rules into the classical DHT model.	[13]
			Release of new identities by CAs.	For the moment, it is the only way to tackle the Sybil Attack effectively.	Introduction of centralization elements into the system; probability of single point of failure.	[7]
Join	False ID creation	Trusted certification	ID obtaining through double-key encryption of newcomer's IP address by a central agent returning ID via SMS.	Significant reduction in the number of sybils.	Single point of failure; different SIMs use possible; PKI elements introduction; IP addresses and phone numbers list; central agents to avoid multiple IDs issue to same peer, additional memory resources and costs.	[20]
			Registration through a centralized authority, list of trusted identities.	Secure bootstrap.	Single point of failure, introducing centralization elements, stolen Ids.	[17]
			Central trusted certification authority assigning and signing *nodeId* certificates that bind ID to a public key and to the IP address.	Randomly chosen *nodeIDs* and ID forging prevented; binding IDs and IP addresses makes it harder for an attacker to move ID across the nodes.	Single point of failure; if an IP address is changed, the corresponding *nodeId* and certificate become invalid.	[14]
		Central. trusted certif.	Periodical invalidation of node IDs by a trusted entity broadcasting a different initialization vector for the hash computations.	It is hard for an attacker to accumulate many node IDs over time and to reuse them.	Legitimate nodes must periodically spend additional time to maintain their membership.	[14]
Join	False ID creation	Distributed trusted certif.	Registration of the ID (IP address and port hash) of a new node by other peers according to IP address prefix; bound to the maximum number of nodes per participant.	External ID usage, no centralization elements; no need to obtain new separate ID; easily applicable to other DHTs besides Chord.	Distributed ID assignment, Sybil-proof only for a limited time; only one node behind a NAT; nodes can obtain a huge number of IPv6 addresses.	[21]
		Costs or fees	Certificates binding the ID to a public key should be paid.	The cost of an attack grows with the size of the network; fees are supposed to fund the operations of the CA.	Single point of failure.	[14]

3.1. Kademlia lookup procedure

In this work we consider both the iterative and the recursive [32] routing of Kademlia. More precisely we consider only iterative Kademlia when we analyze routing algorithms, whereas we examine both when we consider the storage and retrieval process as, in this case, it does not matter whether the Initiator of a lookup request has or not the control over the whole routing process. As a matter of fact, in case the focus is on the storage, the only important thing is to reach a final list of locations for the resource to be stored in or to be retrieved from, whereas in the case of routing it is important, in order to correctly apply our balanced trust-based algorithm, that the Initiator has the control on the local temporary research list at each step of the research. In the iterative routing of Kademlia, the answer of a resource query, launched by an Initiator, is a list of nodes to ask at the next step until the node responsible for the searched resource is found. This list is then sent back to the Initiator at each iterative step by the queried nodes. Therefore, the Initiator is responsible for all the queries in the lookup process and it has also the full control over the whole procedure. On the other hand, in recursive Kademlia, also called R-Kademlia, the routing process is in charge, step by step, of the previously queried node. Only the final list, containing the closest nodes to the target resource key, is sent back to the Initiator that does not have full control over the whole routing process. In the following we briefly recall both the iterative and recursive standard routing and the subsequent standard storage or retrieval process.

Kademlia nodes store contact information to route query messages, keeping lists of (IP address; UDP port; Node ID) triples for nodes at distance from themselves between $2i$ and $2i+1$ and for i ranging from 0 to 160 excluded. These lists are known as $k-buckets$, where k is both the length of the lists and a system parameter chosen such that any given k nodes are very unlikely to fail within an hour of each other (usually $k=20$). When trying to store or retrieve some resources, the requesting peer, the Initiator, runs α parallel asynchronous queries, called *lookups*, for a given resource key at the same time, where α is another system parameter, usually set to 3. The α peers are chosen by the originator peer (we call both the actual Initiator or the intermediate peer in charge of forwarding the lookup request recursively thus) according to a certain policy among those k peers closest to the targeted resource; these peers are stored in a temporary list to be updated by the originator at each step, in the case of iterative Kademlia, or they are simply received by the current in charge peer, in the case of recursive Kademlia [32]. Each of the α peers contacted responds to the querying peer with the k nodes closest to the target resource key according to their peer routing tables. The querying peer can have, as a response, at most $\alpha \times k$ nodes; among these it selects for the next step only the k closest to the searched resource. Both the iterative and the recursive lookup processes end up when at step n, among the at most $\alpha \times k$ responses, the querying peer is unable to find at least one peer closer to the resource than any other already present in the temporary list updated at step $n-1$. At this point the Initiator perform a GET or PUT operation on the closest nodes retrieved. In the case of a PUT operation this also guarantees redundancy, and therefore robustness to the P2P network.

3.2. Simulation conditions

In this section we briefly summarize the main conditions of our simulations and the tool we deployed to study a Kademlia network under a Sybil attack. As a simulator, we employed DEUS [33], a discrete event simulator, where simulation time is sampled in virtual seconds (*vs*). DEUS is developed in Java language by the Distributed System Group of the Information Engineering Department of University of Parma. Among all its capabilities, we obviously focused on its embedded Kademlia implementation, using opportunely modified XML simulation scripts, in order to allow for trust and reputation management. For simplicity's sake, we worked in conditions of steady state network so that, despite nodes joining and leaving, a constant number of nodes, on average, is always present in the network; this is obtained starting from approximately 10,000 *vs* from the start of the simulation. Particularly the number of good nodes, that is, nodes that behave correctly according to the standard Kademlia algorithm, is always 100 on average, whereas the number of bogus nodes, the sybils, varies from simulation to simulation: 0, 2, 10, 50, 100, and 200 on average. 100 and 200 sybils, even if unrealistic, were considered to study the performances of the network and of our balanced trust algorithm in conditions of great stress and in environments with an overwhelming probability of attack. Peers join or leave the network till about 10,000 *vs* according to a periodic process of period 100 *vs* for good nodes and random period for malicious nodes respectively. Other parameters of the Kademlia network are α and k. k is always equal to 20 as suggested in [4], while α, as it influences only routing operations, was fixed to 1 in evaluating storage and retrieval attacks, whereas it is a variable parameter when evaluating routing attacks. The key space size was set to 2000. In DEUS simulation model of Kademlia another parameter is present, namely *discoveryMaxWait*, fixed to 1000 *vs*; this is needed to simulate nodes answering too slowly. In our considerations we will only focus on correctly ended value lookup procedures that start only when the network is steady and with no expiration of the *discoveryMaxWait* parameter; if *discoveryMaxWait* expires the requests are labeled as "unended". Finally, we considered average values over 200 seeds, for each simulation set of parameters, in order to achieve a 94% confidence interval on the results.

In this work we were interested into both routing and storage or retrieval attacks coming from the sybils, that is why we considered various bad behaviors in our simulations: both as a response to a *PUT*/*GET* operation and during the routing of lookup requests. As regards routing, two malevolent behaviors were considered: sybils asked for a next hop can answer back with an empty list or they can respond with a list made of randomly chosen peers. These are only some of the possible malicious behaviors and they were selected as representative of routing attacks and to make some general considerations on Kademlia performances and on the effectiveness of our proposed trust-based solution. Considering storage and retrieval, three malicious behaviors were mainly considered: one regarding a PUT operation and two concerning a GET operation. In the first case the sybils asked of storing a certain resource refuse to do so, in the second case, those sybils requested for retrieving a certain resource

respond with a null value or a corrupted resource (fake resource). It should also be noted that these are only some of the possible malicious behaviors; they were selected as the most representative of a storage and retrieval attack (see Section 2). Furthermore, considering always PUT and GET procedures, we briefly investigated also gray-hole attacks to assess our trust-based scheme in presence of more sophisticated onrushes. As highlighted also in Section 5.2, we took the following misbehaviors into consideration: the acceptance of a PUT and the subsequent refusal to a GET, and a selective refusal of PUTs and GETs on the basis of the target resource.

3.3. Effects of sybils on standard Kademlia

In spite of various metrics, which can be used to evaluate a Sybil attack [34,35], we have concentrated mainly onto the analysis of the following features:

- as regards the *routing process*, the percentage of filling of the k-buckets averaged over all the nodes in the network and the average number of successful lookup procedures;
- as regards the *storage and retrieval*, the average number of successful PUT and GET procedures over all the PUT and GET operations carried out, respectively.

As successful lookups we considered researches arrived correctly at the nodes closest to the target resource in a *find_node* Remote Procedure Call (RPC), whereas for a correct PUT operation we took into account *store* RPCs where all final nodes accepted the storing request. As regards successful GET operations we considered *find_value* RPCs where the retrieved resource is genuine. As incorrect or malevolent behaviors we considered the following:

- for the *routing processes*: nodes responding with a null or a random list;
- for the *storage process*: nodes refusing to accept and store incoming resources for which they should be responsible in the overlay. In this case sybil nodes perform a sort of denial of service attack [36].
- as regards the *retrieval process*: nodes responding with null or not genuine resources.

For our first set of simulations, we focused on the routing process, so in Figs. 2 and 3 we analyzed k-bucket filling and the number of successful lookups. The results depicted in the aforementioned figures refer to sybil nodes responding with a null list and show the dramatic decrease of performances, in both analyzed metrics, as the number of sybils increases. In these graphs the α parameter is fixed to 1 so no parallelism is exploited.

In Fig. 2 the percentage of k-bucket filling is not particularly affected by a small number of sybils (namely 2), and this confirms [37] in saying that there is a minimum percentage of sybils in the routing tables to make a Sybil attack effective. However the filling of k-buckets drops for a number of sybils greater than 50 nodes, reaching a level close to zero when the number of fake IDs is twice the number of the true ones. In Fig. 3 the number of unsuccessful searches is somehow limited considering a number of sybils less than 10, while it significantly increases and overtakes successful searches from

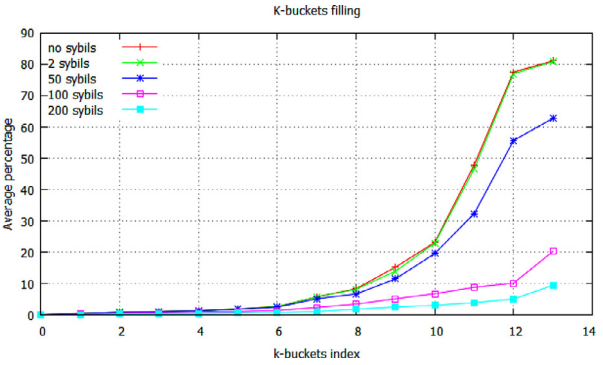


Fig. 2. K-buckets filling versus k-bucket index with variable number of sybils in the network.

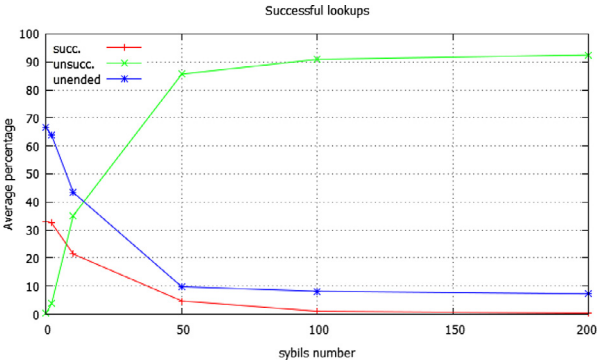


Fig. 3. Successful, unsuccessful and unended lookup procedures with sybils answering with a null list and $\alpha = 1$.

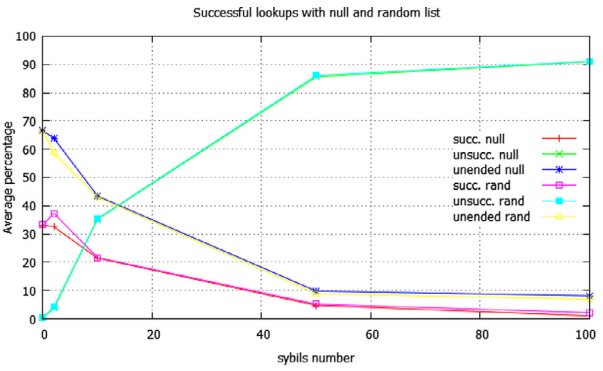


Fig. 4. Comparison between two malicious behaviors in routing process.

10 on. The decrease in not ended searches with an increasing number of sybils is also relevant: probably the more the sybils, the more their ability to divert correct ongoing searching paths to dead points.

In Fig. 4 we compared the performance degradation when the malicious behaviors take place in a look up procedure:

1. the case in which sybils answer with a null list of nodes or
2. with a random list.

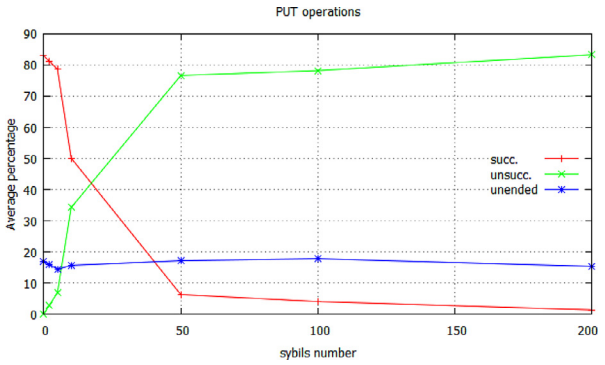


Fig. 5. Successful, unsuccessful and unended PUT procedures with sybils performing a DoS attack.

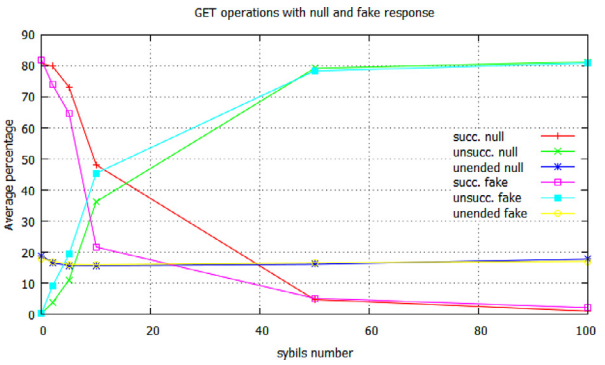


Fig. 6. Comparison between two malicious behaviors in response to a GET request.

From Fig. 4 we can notice that the difference is very slight. This could be explained considering the functioning of Kademlia standard algorithm as recalled in Section 3.1. In case a node answers with a *null* list the first node of the temporary list does not surely update, therefore this behavior forces to go into the searching k nodes phase. From this point the procedure, as described above, could stop or proceed with only one step more than if no malicious nodes where found on the path. The same happens if we consider a malicious node answering with a random list even if in theory there could be the chance for the local list to acquire a new first node anyway. This could explain why the graphs in Fig. 4 are very similar.

As a second set of simulations, we studied the effects of sybil nodes on the PUT and GET procedures. In this case the α parameter was fixed to 1 as it does not influence storage or retrieval but only routing parallelism. The results depicted in Fig. 5 refer to PUT operations, refusing to store the resource. Our results show the impressive decrease of successful storage operations as the number of sybils in the network increases. Anyway the number of unended procedures is almost constant, as we did not considered malicious activities of the sybils in the routing process in this case. Moreover, in Fig. 5, we verified that, also in this case, a small number of sybils (namely 2 or 5) cannot significantly affect the performances, and this confirms the findings in [37] yet again. Anyway the performances decrease, also in this case, for a number of sybils greater than 50 nodes, reaching a level close to zero when the number of fake IDs becomes preponderant.

Given these conditions, the number of unsuccessful PUT requests is somehow limited considering a number of sybils less or equal to 10, while it significantly increases and overtakes successful searches from a certain value, little greater than 10, onwards. The saturation behavior the curves experience when the number of sybils is greater than 50, besides the constancy in not ended operations with an increasing number of sybils, is also relevant. This effect, present also in Fig. 3 for the routing process, can lead to think there is a sort of threshold in the number of sybil nodes, after which the negative effects on the network are ensured. This threshold surely depends on the number of nodes in the network and the relative fractions of good and bad nodes.

In Fig. 6 we studied the difference in performance degradation between the main considered malicious behaviors in a GET procedure:

1. the case in which sybils answer with a null value or
2. with a fake value.

From Fig. 6 we can see that there is a very slight difference in the two cases for a large number or a very little number of sybils in the network. On the contrary, considering a number of sybils ranging from 2 to 10 the difference between the two curves enlarges. This can be explained with the following considerations. In case a node answers with a null value the originator node has the ability to perform a second request on the Kademlia network and therefore, in this second case, it may succeed in reaching a not null resource with a greater probability. In the other case, the originator cannot verify whether the retrieved resource is genuine or not and so it does not perform a further GET request on the network. This leads to a worse degradation of the overall performances, but this effect disappears when the number of sybils becomes extremely preponderant in the network, and the probability of obtaining a genuine resource performing a second GET request drops dramatically. These considerations could explain why the graphs in Fig. 6 are very similar when the number of sybils is very small or very large and why there is a significant difference when their number ranges from 2 to 10.

After these considerations we addressed towards the research of some solutions, or better, some ways to mitigate the effects of malicious nodes on both the LOOKUP and the PUT or GET procedures. With this aim in mind we focused on a solution concerning trust and reputation like in [31] or other solutions present in the literature; this seems a suitable technique in P2P networks, as, like in any human community, nodes interact with each other, create new contacts, and progressively gain their own experience and reputation about each other, but, at our knowledge, this was applied to Kademlia networks only in [8,27,38].

4. S-Kademlia: balanced trust-based DHT algorithm

We propose to improve Kademlia resilience to incorrect operations caused by sybils, introducing trust in the standard Kademlia algorithm. Our approach has been studied from several different viewpoints: we applied trust both to

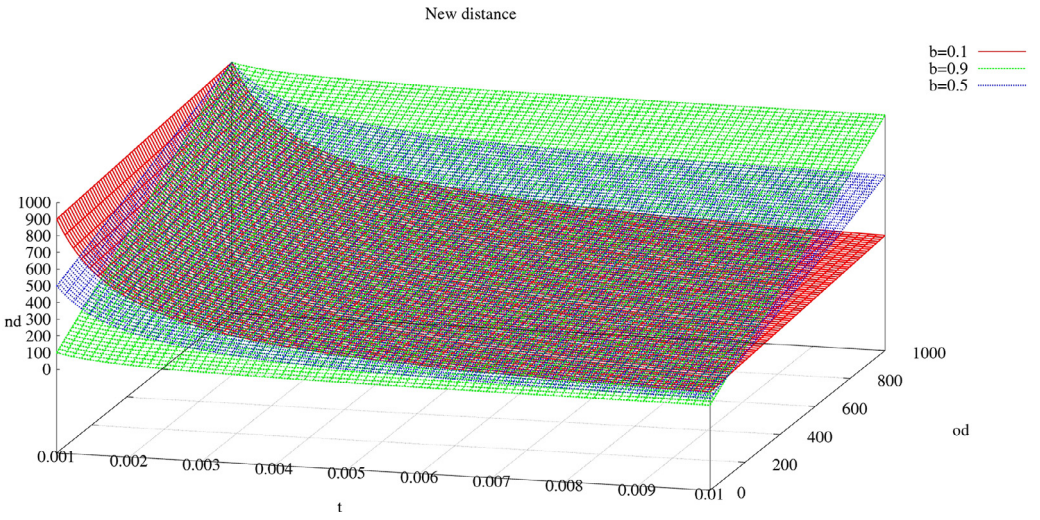


Fig. 7. New distance function.

the iterative routing process and to the storage and retrieval process, in this case independently whether the iterative or recursive scheme is employed. The main concept is taking a trust value into account when ordering the local temporary routing list or the final list of k peers where to store or retrieve the searched resource. Anyway, differently from [8], we thought that the use of a brute force ordering of such lists just on the basis of trust could not work or work very worse than standard Kademlia, maybe protecting from some consequences of a Sybil attack but resulting not convenient. So we tried to define a new metric, with which ordering the aforementioned lists, that takes advantage of some interesting results in load balancing [39,40]. We called it "new distance" (nd), and computed it according to Eq. (1):

$$nd = od \times b + (1 - b) \times \frac{1}{t} \tag{1}$$

where nd is the new distance that taking trust and reputation into account as well, od the old distance computed according to the XOR operation as it is the case with standard Kademlia, b a balancing factor ranging from 0 to 1 and t the reliability factor ranging from a very small number near zero (say ϵ) and infinite. The nodes having the highest new position, or the lowest new distance, which amounts to the same, are chosen as the candidates for storing or retrieving the resource. The function representing the new distance is plotted in Fig. 7 with different values of b. The aforementioned figure shows the two different dependences: linear on the old distance and hyperbolic on trust. The balance factor b can be used to aptly tune the weight of the old distance or of trust.

The new balanced distance, as already stated, can be applied either to the routing or to the storage and retrieval process. In the following we describe the new S-Kademlia algorithm, from both the routing point of view and the storage viewpoint.

Speaking about routing we considered, as already stated, only iterative Kademlia. So at each iterative step the Initiator of a *lookup* chooses the nodes having the smallest new distance as the candidates to be queried for the next hop. The new balanced routing algorithm can be seen throughout the following steps:

- *step 1*: the Initiator of the search, looking for a determined resource, chooses α nodes to query among the k nearest (according to the *old distance*) he knows;
- for the *following steps*, each chosen node undertakes the same procedure. It chooses other α nodes for the next step and sends back to the Initiator their addresses to enable it to contact them. At each step of our balanced algorithm the Initiator orders the local temporary search list with the first position represented by the node with smallest new distance among the k possible ones obtained from one of the α nodes of the previous step. In case the nd values are all the same the choice could be done according to a different policy, e.g., randomly.

Considering storage and retrieval, we considered both iterative and recursive Kademlia. The storing or retrieval algorithm can be described as follows:

- *step 1*: the peer currently in charge of the search, looking for a determined resource to retrieve or to put, gets the final list of peers, closest to the correct position of the resource according to standard Kademlia algorithm. Nodes in this list are ordered according to a certain position according to the *old distance* metric;
- *step 2*: the in charge peer reorders such a list on the basis of its local trust assigned to each peer present in the final list. This reordering is originator-dependent as, in the case of iterative Kademlia, it is the actual Initiator of the *find_value* or *store* RPC to be in charge of reordering the list according to its local perceived trust into other peers, whereas, in the case of recursive Kademlia, it is the final peer in charge to reorder the list before performing the PUT or GET operation. For a PUT operation only peers at position greater than a certain threshold are chosen, in order to guarantee the intrinsic redundancy of Kademlia, whereas for a GET operation only peers at the very first position are elected.

4.1. Trust management

Trust information, in our model, is computed according to the definition found in [31] rather than the one in [27] or [8]. That is, we accounted for an environment risk factor in case no interactions with nodes were previously enacted, in order to avoid the grace phase of [8], and we gave much more importance to direct experience than to recommendations, which are not considered in our model. The trust factor, in its whole form, is an opportune combination of risk and reputation, performed with the contribution of numerical weights. Similarly to PET we considered the trust factor (t) defined as follows:

$$t = W_{R_e} \times R_e + W_{R_r} \times R_r \tag{2}$$

where R_e and R_r are respectively the reputation and the risk, whereas W_{R_e} and W_{R_r} are respectively the weight for the reputation and for the environment risk (values ranging from 0 to 1). We hereby simplified the PET model considering only one bad service level (L = low grade). Our computation of the risk R_r, e.g., the level of vulnerability of the network, is therefore reduced to the following formula:

$$R_r = \frac{N_L h_L}{h_L N_T} = \frac{N_L}{N_T} \tag{3}$$

N_L is always the number of low-grade interactions experienced till now in the network, but it encompasses different behaviors according to the three scenarios we considered:

1. applying trust *only to the routing process*: in this case N_L regards lookup timeout, resource not reached, etc.;
2. applying trust *only to the storage or retrieval process*: in this case N_L considers storage refusal, null retrieved resource, etc.;
3. applying trust *both to the routing and to the storage or retrieval processes*: in this case N_L is made by all the aforementioned malicious responses.

h_L is the PET map function for level L (but as seen in Eq. 3 it is inconsequential in our simplified model), whereas N_T is the total number of requests launched by the peers in the network (only *find_node* for the first case, only *find_value* and *store* for the second case and all possible RPCs for the third case).

The evaluating node stores in a local table different trust values for each node of the P2P network; in case no interactions with a certain node have been yet experienced the value of t is simply represented by the risk part $W_{R_r} \times R_r$; otherwise t also comprises the reputation part that is updated as follows:

1. in case only routing operations are considered: $+\frac{1}{N_p}$ to each node present in paths leading to the correct resource and $-\frac{2}{N_p}$ to the nodes of incorrect paths,.
2. in case only storage or retrieval operations are considered: $+\frac{1}{N_p}$ to each peer accepting a PUT or responding with a correct resource and $-\frac{2}{N_p}$ to the peers denying a PUT or responding to a GET request with fake or null resources.
3. should the trust be applied to all Kademlia RPCs, t update is given by all the aforementioned considerations.

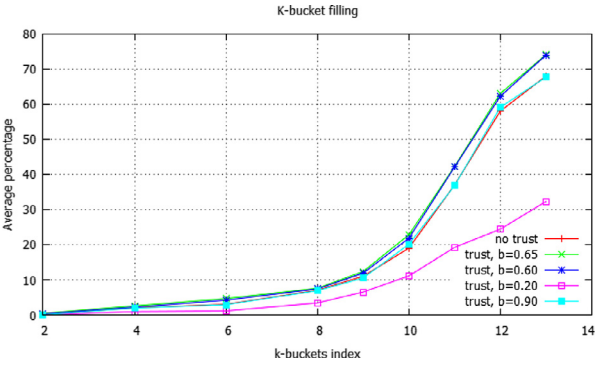


Fig. 8. K-buckets filling versus k-bucket index for trust and no trust algorithms for 50 sybils network and variable value of balancing factor.

N_p is the total number of requests (LOOKUPs in the first case, PUTs or GETs in the second case and all in the third case) received by the considered peer. We do not consider unchoking but, similarly to [8], trust values expire after 24 h in order to avoid that a malicious node may gain too much trust through collusion operations. As regards the routing part, the negative reputation update may concern only the last node, or a limited subset of nodes in the path anyway, in order to take into consideration the possibility that not all the nodes in a wrong path behaved maliciously.

5. Simulation results

In this section we present the results of our analysis of the performances of our new balanced trust-based approach over Kademlia with sybils. The majority of the parameters and of the simulation conditions are the same as the ones in Section 3.2. The value for the parameter b is critical and very important to opportunely tune the performances of our balanced trust-based algorithm. Its optimum value, obtained after a great simulation campaign and averaged over the malicious behaviors considered, is 0.65 as regards only routing operations, whereas it is 0.55 when we considered only storage and retrieval operations. In both situations W_{R_r} was set to 1 for a node joining for the first time and 0.5 for an already joined node, whereas W_{R_e} is 0 for new joining nodes and 0.5 for already joined ones. R_r is positive for new nodes that join the network. In these conditions we analyzed the routing and the storage and retrieval processes separately with our balanced trust-based algorithm.

5.1. Trust applied to routing processes

Considering routing operations, in Fig. 8 the k-bucket filling versus k-bucket index is analyzed considering 50 sybils in the network. Several values of the balancing parameter b were used and, as it can be inferred from the aforementioned figure, the optimum value is about 0.65, whereas a value of b closer to 1 leads the performances close to the ones of a network infected by sybils with no protection and a value of b close to 0 degrades the performances even worse than the presence of sybils. In the latter, case the results can be explained as the routing is done mainly according to the trust

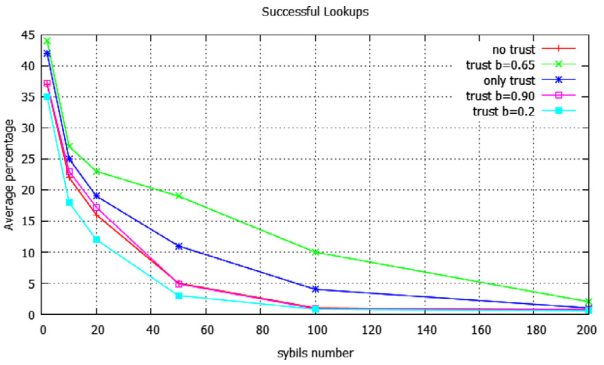


Fig. 9. Comparison of successful lookups for standard Kademlia, only trust method and our balanced routing method with variable balancing factor.

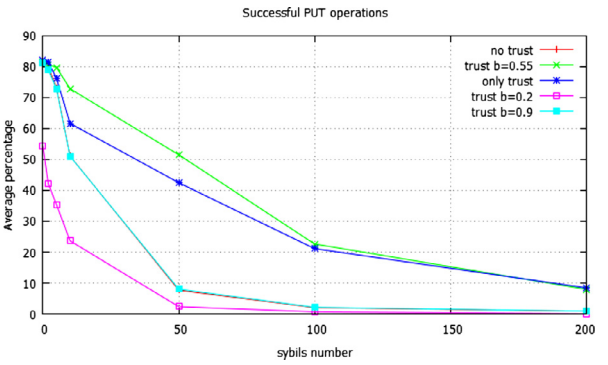


Fig. 10. Comparison of successful PUTs for standard Kademlia, only trust method and our balanced trust method.

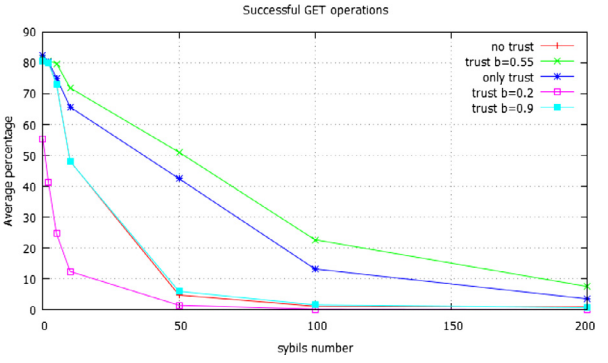


Fig. 11. Comparison of successful GETs for standard Kademlia, only trust method and our balanced trust method.

metric rather than the standard Kademlia algorithm, which is the optimum.

On the other hand, in presence of a relevant number of sybils, our approach mitigates their negative effect even, albeit not reaching the performances of standard Kademlia algorithm with no sybils. This is evident also in Fig. 9 where we see how our balanced routing partly alleviates the negative effects of sybils on successful lookups, especially when the number of malicious nodes is less than 200. In the same figure we can see that a not optimized value of b may even help sybils in degrading the performances as it happened in the k-bucket filling. Moreover, always in Fig. 9, we compare our approach with the one described in [8] or [27] (labeled as "*only trust*") showing that a balanced approach is better than a pure trust scheme.

5.2. Trust applied only to storage and retrieval processes

As regards storage and retrieval operations we ran different sets of simulations for PUT and GET operations and for iterative and recursive Kademlia. The average optimum value for the b parameter is in this case slightly smaller than in the routing process, namely 0.55. This value was achieved making an average, over a great simulation campaign, of the main considered malicious behaviors: denying to store a resource and responding with a null or fake resource. Another important finding is that its value is independent from considering iterative or recursive Kademlia routing algorithm.

In Fig. 10 we see how our balanced storing algorithm is effective in the case of PUT operations as well, also in this case when the number of malicious nodes is far less than 200. Moreover, in the same figure, we compare our approach with the one described in [27] showing that a balanced strategy is better than a pure trust method in this case as well, especially when the number of sybils in the network is not preponderant. In this figure we considered successful a PUT request not denied but also correctly assigned to the node being theoretically right in charge. This is the reason why a preponderant value of trust leads to unwanted results and so to unsuccessful PUT operations.

Similar considerations can be drawn from Fig. 11, where we compared our approach and the only trust-based one in the case of GET operations with sybils responding with a fake resource, itself the worst scenario as explained in Section 3.3. Furthermore, we noticed that, differently from the PUT operations, a better performance of our algorithm, compared to the only trust-based one, occurred also in presence of a great amount of sybils within the network.

Moreover, in this scenario, we tried to investigate some more sophisticated types of attack coming from the sybils further. In particular, the gray-hole behaviors considered are the following:

1. accepting a resource so that the PUT is successful and then refusing retrieval operations on that resource;
2. blocking the PUTs and GETs only as regards particular contents or resources.

The first behavior, intuitively, does not affect the success of PUT procedures so we focused on successful GET operations. In this case the performances of our trust-based approach may decrease as shown in Fig. 12, but it strongly depends on the number of retrievals compared with the number of PUTs, and on the number of sybils in the network. Considering r as the ratio of PUTs over GETs, in a fixed time interval (namely 10,000 *vs* in our simulations), we can notice from the analyses that its variation may lead to worse performances for our balanced algorithm. This occurs mainly when r is greater than 1, i.e., the number of PUTs overtakes the number of GETs, and with a number of sybils greater than 10. When the value of r is less than 1, the degradation in the performances of our algorithm starts later, when the number of sybils in the network gets greater than 50. Finally, we

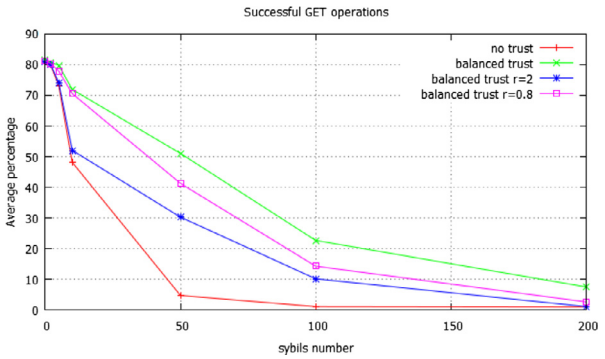


Fig. 12. Successful GETs according to standard algorithm and balanced trust algorithm and variable ratio r between PUTs and GETS, with b = 0.55 and gray-hole type 1 attacks.

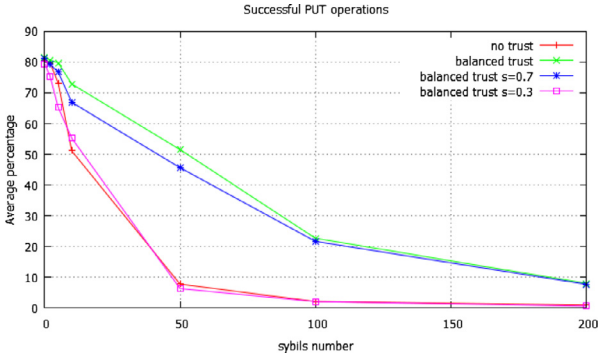


Fig. 13. Successful PUTs according to standard algorithm and balanced trust algorithm and variable popularity ratio s, with b = 0.55 and gray-hole type 2 attacks.

observe that when the number of sybils is not preponderant, the value of r is not so significant and the performances are similar to the case when no sophisticated attacks are enacted. In this situation, a possible way to soothe the attack may be to change a little the trust management: for example updating the trust value with $-\frac{3}{N_p}$ for nodes accepting a PUT and afterwards denying the possession of the target resource.

The second gray-hole behavior, always concerning storage and retrieval operations, may affect both PUTs and GETs. Anyway, the effectiveness of such an attack is to evaluate on the basis of the diffusion of the considered content. If the content is very popular, the attack may be thwarted through our proposed balanced trust-based algorithm with no changes, whereas in case the target resources are not so much widespread, the performances of our algorithm for the target resources decrease. This can be clearly seen in Fig. 13, where successful PUTs are depicted in the aforementioned cases. The spreading factor s is a measure of the popularity of the target resource, the only one considered for PUT procedures in those curves where s is specified in the key. s is defined as the ratio between the PUTs for the target resource and the overall storage operations. As we can see, when the value of s is greater than 0.5 (therefore indicating a very popular resource) the curve is very similar to our standard balanced trust-based algorithm, whereas when s is less than 0.5 the curve experiences an irregular behavior, being both

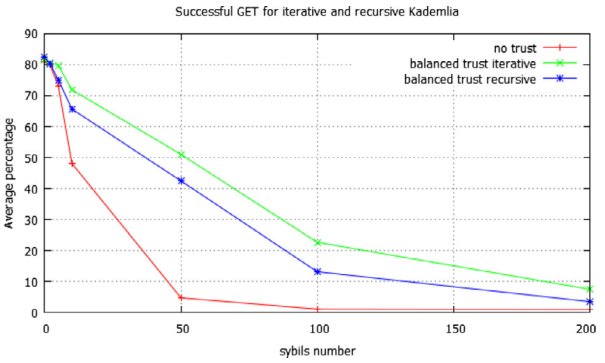


Fig. 14. Successful GET operations in Kademlia according to standard algorithm and balanced trust algorithm (both iterative and recursive) with b = 0.55.

below and above Kademlia standard algorithm, depending on the number of sybils in the network. A possible adaptation of the trust management could be useful under this attack, in order to correctly perform storage or retrieval for the target resource. This could affect both the risk evaluation, considering a further parameter, besides N_L, to take into account content specific bad behaviors, and the trust value update. Anyway an attacker should carefully consider whether to perform such a targeted attack or not. As a matter of fact, launching such a specific bad behavior may affect the effectiveness of the overall attack, as other PUT procedures are not compromised and performing a different attack for all possible contents may result too expensive in terms of computational resources.

Finally, always considering this scenario where trust is applied only to storage and retrieval operations, we were able to compare the iterative and recursive Kademlia versions as well. This is depicted in Fig. 14 where we analyzed our balanced trust approach in both versions of Kademlia, as regards the number of successful GET operations. Performances of standard Kademlia, both in the recursive and in the iterative case, are practically the same and so in the graph they are represented through a single line labeled 'no trust'. Differently, the performances using our trust-based balanced algorithm for storing resources in recursive Kademlia seems slightly outperforming iterative Kademlia. This could be explained as, in our implementation of recursive Kademlia, the final in charge peer sends back, to all the contacted nodes, information regarding the success of the PUTs or the GETs, whereas in iterative Kademlia this information is available only to the actual Initiator of GET or PUT requests. This could lead to a better spreading of trust information among peers.

5.3. Trust applied to all Kademlia RPCs

Finally, we assessed the effectiveness of our balanced trust-based algorithm on the overall Kademlia algorithm. To reach this goal, we focused only on iterative Kademlia, in order to apply trust both to the routing process and to the subsequent storage or retrieval procedure. In these analysis conditions we focused on the percentage of overall successful RPCs, i.e., we took the number of successful lookup requests followed by a PUT or GET operation into account. A

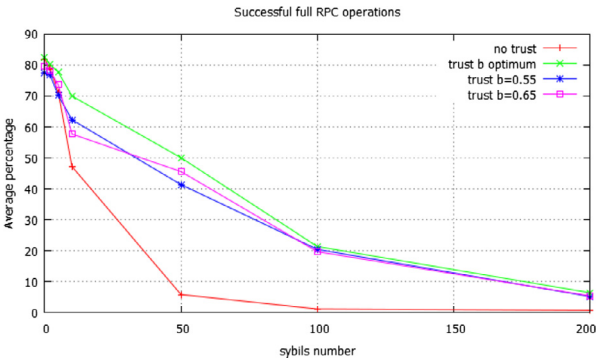


Fig. 15. Successful RPC operations in Kademlia according to standard algorithm and balanced trust algorithm with variable balancing factor.

successful request can be so deemed if it reaches the node responsible for a certain resource, according to standard Kademlia XOR distance, and it succeeds into storing in it the resource or to get from it a not fake resource. The main finding in this set of simulations is that an optimum value for the *b* parameter does not exist, but the maximum of the performances is achieved with a value, ranging from 0.55 to 0.65, which act, respectively, as lower and upper bound. The value of the balancing factor is not simple the mean value but it varies according to the total number of storage or routing operations launched in the network. The main results are depicted in Fig. 15. Obviously, the metrics analyzed in the previous subsections undergo worse performances, separately considered, however they perform better than in the case no trust is deployed.

6. Conclusion

In this paper we carried out a deep analysis of anti-sybil countermeasures present in the literature of the last years and we quantitatively analyzed the effects of a Sybil attack on some important features of a Kademlia network, namely routing and storage and retrieval operations. We presented a combined trust-based algorithm to make the *find_node*, *find_value* and *store* RPCs of a Kademlia network more resilient in presence of malicious nodes with many false identities, considering both some simple malicious behaviors and some more sophisticated attitudes. Our approach, based both on the standard Kademlia algorithm and on a trust-based algorithm, is not centralized and uses a new distance metric trying to vary the possible spectrum of nodes in order to mitigate the malicious behavior without the need for certificates, cryptography or Certification Authorities. The results of our simulations confirm that using a balanced trust algorithm may lead to better results when compared with standard Kademlia and that its performances are better even when compared with those methods using exclusively trust. Future works may consider how to effectively thwart the more sophisticated attacks highlighted in the paper, perhaps exploiting the suggested hints in the trust management, and finding an apt threshold after which the Sybil attack is ensured. Another future research topic could be analyzing the trust information spreading among peers in the recursive version of Kademlia or the variation of the optimum value for the balancing factor in different situations and levels of environmental risk in detail. Finally, finding a mathematical close form for the variation of the optimum value of the balancing factor according to the number of enacted routing or storage and retrieval operations could be very interesting.

Acknowledgments

The author would like to thank his colleague Luca Veltri, who provided insight and expertise that greatly assisted the research, and Antonio Enrico Buonocore for his precious support in proof reading the article.

References

[1] R. Pecori, L. Veltri, A key agreement protocol for P2P VoIP applications, Proceedings of the Seventeenth International Conference on Software, Telecommunications and Computer Networks, Hvar-Korcula-Split, Croatia, September 2009.
[2] R. Pecori, A PKI-free key agreement protocol for P2P VoIP applications, in: Proceedings of the IEEE International Conference on Communications (ICC), Ottawa, Canada, June 2012.
[3] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for internet applications, in: Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, San Diego, CA, USA, August 27th–31st, 2001.
[4] P. Maymounkov, D. Mazires, Kademlia: A Peer-to-peer Information System Based on the XOR Metric, in: Proceedings of the First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 2002.
[5] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer system, in: Proceedings of the Eighteenth IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, November 2001.
[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, in: Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, San Diego, California, USA, August 2001.
[7] J.R. Douceur, The Sybil Attack, in: Proceedings of the IPTPS First International Workshop, Cambridge, MA, USA, March 2002.
[8] M. Kohnen, Analysis and optimization of routing trust values in a Kademlia-based distributed hash table in a malicious environment, in: Proceedings of the Second Baltic Congress on Future Internet Communications (BCFIC), Vilnius, Lithuania, 25th–27th April, 2012.
[9] A. Singh, T.W. Ngan, P. Druschel, D.S. Wallach, Eclipse attack on overlay networks: Threats and defenses, in: Proceedings of the INFOCOM Twenty-fifth IEEE Conference on Computer Communications, Barcelona, Spain, April 2006.
[10] E. Sit, R. Morris, Security considerations for peer-to-peer distributed hash tables, Lecture Notes In Computer Science, vol. 2429.
[11] R. Pecori, L. Veltri, Trust-based routing for Kademlia in a sybil scenario, in: Proceedings of the Twenty-second International Conference on Software, Telecommunications and Computer Networks (SoftCOM), September 17th–19th 2014, Split, Croatia, 2014.
[12] B.N. Levine, C. Shields, N.B. Margolin, A Survey of Solutions to the Sybil Attack Technical report, University of Massachusetts, 2006.
[13] G. Danezis, C. Lesniewski-Laas, M.F. Kaashoek, R. Anderson, Sybil-resistant DHT routing, in: Proceedings of the Tenth European Symposium on Research in Computer Security (ESORICS), September 12th–14th, 2005, Milan, Italy, 2005, pp. 305–318.
[14] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, D.S. Wallach, Secure routing for structured peer-to-peer overlay networks, in: Proceedings of Fifth Usenix Symposium on Operating Systems Design and Implementation, Boston, MA, USA, December 2002.
[15] H. Yu, M. Kaminsky, P.B. Gibbons, A. Flaxman, SybilGuard: Defending against Sybil attacks via social networks, in: Proceedings of the IEEE/ACM Transactions on Networking, Pisa, Italy, September 11th–15th, 2006.
[16] T. Condie, V. Kacholia, S. Sankararaman, J.M. Hellerstein, P. Maniatis, Induced Churn as Shelter from Routing-Table Poisoning, in: Proceedings of the Thirteen Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 2006.
[17] J. Newsome, E. Shi, D. Song, A. Perrig, The Sybil attack in sensor networks: Analysis and defenses, in: Proceedings of the Third Symposium on Information Processing in Sensor Networks (IPSN), Berkeley, CA, USA, April 26th–27th, 2004.

Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

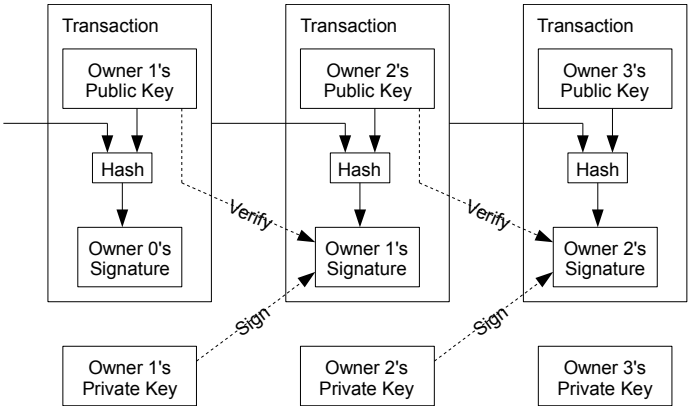
1. Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. Transactions that are computationally impractical to reverse would protect sellers from fraud, and routine escrow mechanisms could easily be implemented to protect buyers. In this paper, we propose a solution to the double-spending problem using a peer-to-peer distributed timestamp server to generate computational proof of the chronological order of transactions. The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.

2. Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.

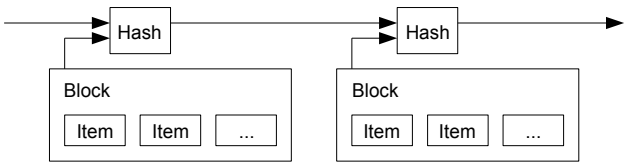


The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint, that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double-spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank.

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced [1], and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received.

3. Timestamp Server

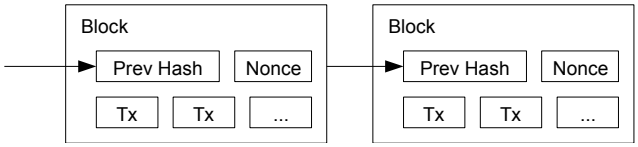
The solution we propose begins with a timestamp server. A timestamp server works by taking a hash of a block of items to be timestamped and widely publishing the hash, such as in a newspaper or Usenet post [2-5]. The timestamp proves that the data must have existed at the time, obviously, in order to get into the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it.



4. Proof-of-Work

To implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof-of-work system similar to Adam Back's Hashcash [6], rather than newspaper or Usenet posts. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash.

For our timestamp network, we implement the proof-of-work by incrementing a nonce in the block until a value is found that gives the block's hash the required zero bits. Once the CPU effort has been expended to make it satisfy the proof-of-work, the block cannot be changed without redoing the work. As later blocks are chained after it, the work to change the block would include redoing all the blocks after it.



The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes. We will show later that the probability of a slower attacker catching up diminishes exponentially as subsequent blocks are added.

To compensate for increasing hardware speed and varying interest in running nodes over time, the proof-of-work difficulty is determined by a moving average targeting an average number of blocks per hour. If they're generated too fast, the difficulty increases.

5. Network

The steps to run the network are as follows:

1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Nodes always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next proof-of-work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one.

New transaction broadcasts do not necessarily need to reach all nodes. As long as they reach many nodes, they will get into a block before long. Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

6. Incentive

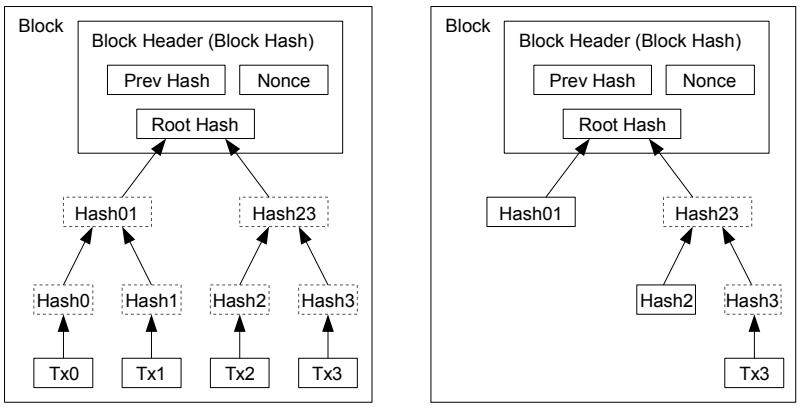
By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.

The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free.

The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable to play by the rules, such rules that favour him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth.

7. Reclaiming Disk Space

Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree [7][2][5], with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored.



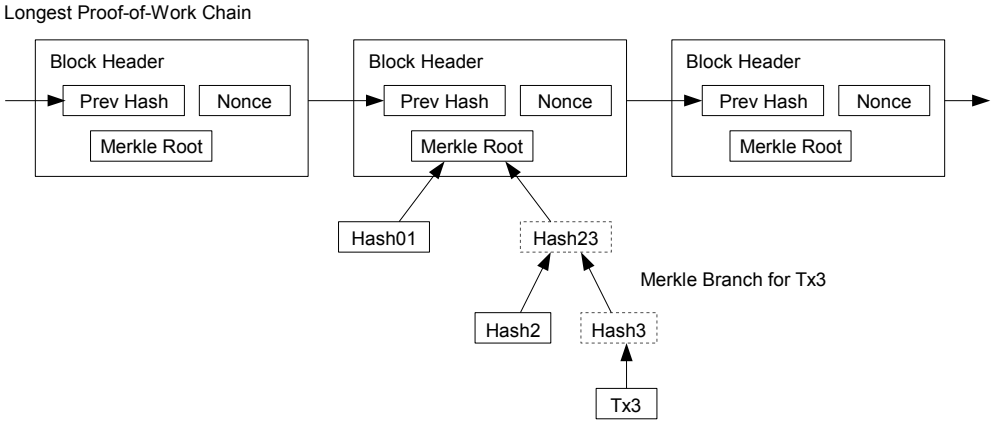
Transactions Hashed in a Merkle Tree

After Pruning Tx0-2 from the Block

A block header with no transactions would be about 80 bytes. If we suppose blocks are generated every 10 minutes, 80 bytes * 6 * 24 * 365 = 4.2MB per year. With computer systems typically selling with 2GB of RAM as of 2008, and Moore's Law predicting current growth of 1.2GB per year, storage should not be a problem even if the block headers must be kept in memory.

8. Simplified Payment Verification

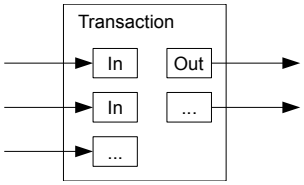
It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.



As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network. One strategy to protect against this would be to accept alerts from network nodes when they detect an invalid block, prompting the user's software to download the full block and alerted transactions to confirm the inconsistency. Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.

9. Combining and Splitting Value

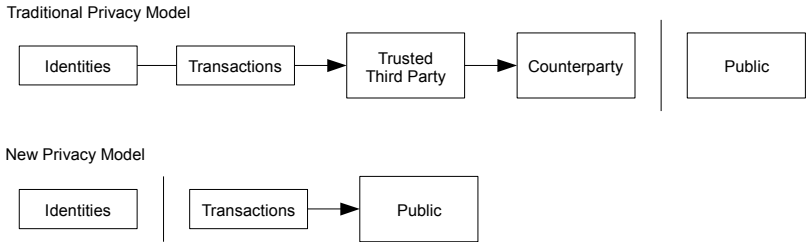
Although it would be possible to handle coins individually, it would be unwieldy to make a separate transaction for every cent in a transfer. To allow value to be split and combined, transactions contain multiple inputs and outputs. Normally there will be either a single input from a larger previous transaction or multiple inputs combining smaller amounts, and at most two outputs: one for the payment, and one returning the change, if any, back to the sender.



It should be noted that fan-out, where a transaction depends on several transactions, and those transactions depend on many more, is not a problem here. There is never the need to extract a complete standalone copy of a transaction's history.

10. Privacy

The traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party. The necessity to announce all transactions publicly precludes this method, but privacy can still be maintained by breaking the flow of information in another place: by keeping public keys anonymous. The public can see that someone is sending an amount to someone else, but without information linking the transaction to anyone. This is similar to the level of information released by stock exchanges, where the time and size of individual trades, the "tape", is made public, but without telling who the parties were.



As an additional firewall, a new key pair should be used for each transaction to keep them from being linked to a common owner. Some linking is still unavoidable with multi-input transactions, which necessarily reveal that their inputs were owned by the same owner. The risk is that if the owner of a key is revealed, linking could reveal other transactions that belonged to the same owner.

11. Calculations

We consider the scenario of an attacker trying to generate an alternate chain faster than the honest chain. Even if this is accomplished, it does not throw the system open to arbitrary changes, such as creating value out of thin air or taking money that never belonged to the attacker. Nodes are not going to accept an invalid transaction as payment, and honest nodes will never accept a block containing them. An attacker can only try to change one of his own transactions to take back money he recently spent.

The race between the honest chain and an attacker chain can be characterized as a Binomial Random Walk. The success event is the honest chain being extended by one block, increasing its lead by +1, and the failure event is the attacker's chain being extended by one block, reducing the gap by -1.

The probability of an attacker catching up from a given deficit is analogous to a Gambler's Ruin problem. Suppose a gambler with unlimited credit starts at a deficit and plays potentially an infinite number of trials to try to reach breakeven. We can calculate the probability he ever reaches breakeven, or that an attacker ever catches up with the honest chain, as follows [8]:

p = probability an honest node finds the next block
q = probability the attacker finds the next block
q_z = probability the attacker will ever catch up from z blocks behind

$$q_z = \begin{cases} 1 & \text{if } p \le q \\ (q/p)^z & \text{if } p > q \end{cases}$$

Given our assumption that $p > q$, the probability drops exponentially as the number of blocks the attacker has to catch up with increases. With the odds against him, if he doesn't make a lucky lunge forward early on, his chances become vanishingly small as he falls further behind.

We now consider how long the recipient of a new transaction needs to wait before being sufficiently certain the sender can't change the transaction. We assume the sender is an attacker who wants to make the recipient believe he paid him for a while, then switch it to pay back to himself after some time has passed. The receiver will be alerted when that happens, but the sender hopes it will be too late.

The receiver generates a new key pair and gives the public key to the sender shortly before signing. This prevents the sender from preparing a chain of blocks ahead of time by working on it continuously until he is lucky enough to get far enough ahead, then executing the transaction at that moment. Once the transaction is sent, the dishonest sender starts working in secret on a parallel chain containing an alternate version of his transaction.

The recipient waits until the transaction has been added to a block and z blocks have been linked after it. He doesn't know the exact amount of progress the attacker has made, but assuming the honest blocks took the average expected time per block, the attacker's potential progress will be a Poisson distribution with expected value:

$$\lambda = z\frac{q}{p}$$

To get the probability the attacker could still catch up now, we multiply the Poisson density for each amount of progress he could have made by the probability he could catch up from that point:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{cases}$$

Rearranging to avoid summing the infinite tail of the distribution...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

Converting to C code...

```
#include <math.h>
double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

Running some results, we can see the probability drop off exponentially with z.

```
q=0.1
z=0     P=1.0000000
z=1     P=0.2045873
z=2     P=0.0509779
z=3     P=0.0131722
z=4     P=0.0034552
z=5     P=0.0009137
z=6     P=0.0002428
z=7     P=0.0000647
z=8     P=0.0000173
z=9     P=0.0000046
z=10    P=0.0000012

q=0.3
z=0     P=1.0000000
z=5     P=0.1773523
z=10    P=0.0416605
z=15    P=0.0101008
z=20    P=0.0024804
z=25    P=0.0006132
z=30    P=0.0001522
z=35    P=0.0000379
z=40    P=0.0000095
z=45    P=0.0000024
z=50    P=0.0000006
```

Solving for P less than 0.1%...

```
P < 0.001
q=0.10   z=5
q=0.15   z=8
q=0.20   z=11
q=0.25   z=15
q=0.30   z=24
q=0.35   z=41
q=0.40   z=89
q=0.45   z=340
```

12. Conclusion

We have proposed a system for electronic transactions without relying on trust. We started with the usual framework of coins made from digital signatures, which provides strong control of ownership, but is incomplete without a way to prevent double-spending. To solve this, we proposed a peer-to-peer network using proof-of-work to record a public history of transactions that quickly becomes computationally impractical for an attacker to change if honest nodes control a majority of CPU power. The network is robust in its unstructured simplicity. Nodes work all at once with little coordination. They do not need to be identified, since messages are not routed to any particular place and only need to be delivered on a best effort basis. Nodes can leave and rejoin the network at will, accepting the proof-of-work chain as proof of what happened while they were gone. They vote with their CPU power, expressing their acceptance of valid blocks by working on extending them and rejecting invalid blocks by refusing to work on them. Any needed rules and incentives can be enforced with this consensus mechanism.

RepuCoin: Your Reputation Is Your Power

Jiangshan Yu, David Kozhaya, Jeremie Decouchant, and Paulo Esteves-Verissimo, *Fellow, IEEE*

Abstract—Existing proof-of-work cryptocurrencies cannot tolerate attackers controlling more than 50 percent of the network's computing power *at any time*, but assume that such a condition happening is "unlikely". However, recent attack sophistication, e.g., where attackers can rent mining capacity to obtain a majority of computing power temporarily, render this assumption unrealistic. This paper proposes RepuCoin, the first system to provide guarantees even when more than 50 percent of the system's computing power is temporarily dominated by an attacker. RepuCoin physically limits the rate of voting power growth of the entire system. In particular, RepuCoin defines a miner's power by its 'reputation', as a function of its work integrated over the time of the entire blockchain, rather than through instantaneous computing power, which can be obtained relatively quickly and/or temporarily. As an example, after a single year of operation, RepuCoin can tolerate attacks compromising 51 percent of the network's computing resources, even if such power stays maliciously seized for almost a whole year. Moreover, RepuCoin provides better resilience to known attacks, compared to existing proof-of-work systems, while achieving a high throughput of 10000 transactions per second (TPS).

Index Terms—Blockchain, cryptocurrency, fault tolerance, consensus

1 Introduction

Bitcoin [1] is the most successful decentralized cryptocurrency to date. However, despite its enormous commercial success, many weaknesses have been associated with Bitcoin, including weak consistency, low transaction throughput and vulnerabilities to attacks, such as double spending attacks [2], [3], eclipse attacks [4], [5], selfish-mining attacks [6], [7], and flash attacks [8].

Several promising existing solutions [9], [10], [11], [12], [13], [14] targeted the low throughput problem of Bitcoin. Nevertheless, these solutions either provide only probabilistic guarantees about transactions (weak consistency) [13] or can provide strong consistency but suffer from liveness problems even when an attacker has a relatively small computing power [15]. Moreover, the resilience of such solutions against attacks, such as selfish mining attacks where an attacker has more than 25 percent of the computing power [3], [6], [7], [16], remains unsatisfactory. In addition, all existing contemporary proof-of-work (PoW) based variants of Bitcoin (e.g., [13], [15], [17], [18]) rely on the assumption that an attacker cannot have more than 33 or 50 percent of computing power *at any time*. However, with the sophistication of attacks mounted on Bitcoin, e.g., flash attacks (a.k.a. bribery attacks), where an attacker can obtain a temporary majority (> 50 percent) of computing power by renting mining capacity [8], all these systems would fail. In brief, existing solutions that address the weaknesses associated with Bitcoin still suffer from significant shortcomings.

This paper addresses these shortcomings —liveness of current high-throughput solutions, and vulnerability to attacks such as selfish mining and flash attacks. In particular, we propose RepuCoin, the first system that can prevent attacks against an attacker who may possess more than 50 percent computing power of the entire network temporarily (e.g., a few weeks or even months). Our proof-of-concept implementation shows that while providing better security guarantees than predecessor protocols, RepuCoin also ensures a very high throughput (10000 transactions per second). In practice, Visa confirms a transaction within seconds, and processes 1.7k TPS on average [19]. This shows that RepuCoin satisfies the required throughput of real world applications.

Design Principle. Our system addresses the aforementioned challenges by defining a new design principle, called *proof-of-reputation*. *Proof-of-reputation* is based on proof-of-work, but with two fundamental improvements.

First, under proof-of-reputation a miner's decision power (i.e., the voting power for reaching consensus in the system) is given by its reputation. A miner's reputation is not measured by what we call the miner's 'instantaneous' power, i.e., the miner's computing power in a short time range, as in classic PoW. Instead, the reputation is computed based on both the *total amount of valid work* a miner has contributed to the system and the *regularity of that work, over the entire period of time* during which the system has been active. We call this the miner's 'integrated power'. So, when an attacker joins the system at time t, even if it has a very strong mining ability that is, high computational (i.e., instantaneous) power, it would have no integrated power at time t, or even shortly after, as it did not contribute to the system before t.

Second, when a miner deviates from the system specifications, RepuCoin lowers the miner's reputation, and hence

- *J. Yu is with Faculty of Information Technology, Monash University, Melbourne, Clayton VIC 3800, Australia. E-mail: j.yu.research@gmail.com.*
- *D. Kozhaya is with ABB Corporate Research, Baden-Dattwil CH-5405, Switzerland. E-mail: david.kozhaya@gmail.com.*
- *J. Decouchant and P. Esteves-Verissimo are with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT), University of Luxembourg, Esch-sur-Alzette 4365, Luxembourg. E-mail: {jeremie.decouchant, paulo.verissimo}@uni.lu.*

Manuscript received 18 Sept. 2018; revised 24 Jan. 2019; accepted 31 Jan. 2019. Date of publication 19 Feb. 2019; date of current version 17 July 2019. (Corresponding author: Jiangshan Yu.)
Recommended for acceptance by J. D. Bruguera.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2019.2900648

its integrated power, in consequence of this negative contribution. This prevents a powerful malicious miner from attacking the system repeatedly without significant consequences. In contrast, classic PoW systems either do not support any feature for punishing miners that do not abide by system specifications, or they punish these miners by merely revoking their rewards—this does not prevent them from attacking the system again immediately after.

RepuCoin provides deterministic guarantees on transactions by employing a *reputation-based weighted voting consensus*. Consensus is carried by a group formed of the top reputable miners. Every member of that group has a weight associated to its vote. The weight of a member's vote is the percentage of that member's reputation w.r.t. the entire group's reputation. Such weights ensure that one's voting power depends, not on the sheer instantaneous (computing) power—which is the enabler of flash attacks— but on the integrated power, which both takes time to build, and is built on the miner's honesty and historical performance. In fact, quantifying a miner's voting power based on its performance in the entire blockchain highlights the self-stabilizing characteristics of our approach: qualitatively, to acquire power in RepuCoin a miner is urged to exhibit normal, honest behavior; quantitatively, the speed with which a miner can gain power is dictated by the regularity and amount of that miner's contributions in the entire blockchain.

To illustrate the robustness of the design choices underlying RepuCoin, we present our analysis of the security provided by the mechanisms used in RepuCoin. In particular, we show that achieving the safety and liveness correctness conditions of the reputation-based weighted voting consensus protocol is physically guaranteed by the growth rate of the proof-of-reputation function, and the rate of decision power growth of the entire system is bounded. In addition, we present experiments exemplifying concrete values for the decision power growth in several situations, showing that the network achieves very high stochastic robustness against attacks on its liveness or safety. For instance, we demonstrate that after a single year of operation, RepuCoin is resilient to all attacks that compromise 26, 33, and 51 percent of the networks computing resources, even if such power stays maliciously confiscated for almost 100 years, 2 years, and 1 year respectively. Also, in the same setting, even if an attacker can afford to seize a huge computational power for a specific period, due to the cost of such attacks (e.g., 90 percent for up to 3 months), it will not break RepuCoin. Moreover, we provide an analysis of the non-rationality of infiltration attacks, with a comparison of the cost of attacking different systems. Furthermore, we provide in detail how RepuCoin prevents known attacks.

2 Related Work

Consensus is the key component and backbone of Blockchains, which use two main types of consensus mechanisms, namely proof-of-X based consensus, and Byzantine-fault tolerant (BFT) consensus. The former is generally permissionless, where anyone can join and leave the potentially large consensus group; and the latter is permissioned, where the set of participants running consensus is small and predefined. Due to space limitations, we give a brief review here, and refer readers to our report [20], and the associated works for more details.

Proof-of-X based consensus has gained much interest since the use of Proof-of-work in Bitcoin, already described in the introduction. Proof-of-stake, which was first discussed in a Bitcoin community forum [21], has been proposed to use virtual voting to provide quicker transaction confirmation. Proof-of-space (a.k.a. proof-of-capacity) [22], [23] has been proposed to use physical storage resources to replace computing power in the proof-of-work mechanism. Proof-of-coin-age [24] shares a similar concept as proof-of-stake, as participants also perform virtual "mining" by demonstrating possession of a quantity of currency. Proof-of-activity [25] puts every coin owner into a mining lottery; periodically, winners are randomly determined by transactions. A winner is expected to respond with a signed message within a small time interval to claim its award. Proof-of-elapsed time [26], proposed by Intel and implemented as a Hyperledger project, uses Intel SGX enabled CPUs to do virtual voting through a random sleeping time to replace the proof-of-work mechanism. Proof-of-membership system in ByzCoin [15] creates a consensus group formed by recent PoW-based block creators, and this group runs a BFT protocol for reaching consensus. However, it is shown that ByzCoin still suffers from selfish mining attack [15], and that it can permanently lose liveness during reconfiguration, if too many miners disappear in a short time [27] or can repeatedly lose liveness temporarily [15]. Unfortunately, none of the permissionless consensus protocols described is able to provide a meaningful security guarantee when an attacker is able to control a majority of mining power.

Notable state of the art of BFT protocols include PBFT [28], MinBFT/MinZyzzyva [29], or ReBFT [30]. PBFT [28] proposes the first Byzantine fault-tolerant algorithm that has an acceptable performance in practice. MinBFT/MinZyzzyva [29] are hybrid protocols presenting non-speculative and speculative BFT algorithms, which are efficient and only require $2f + 1$ replicas (rather than $3f + 1$) in total to tolerate a maximum number f of faulty players, thanks to the use of a trusted monotonic counter associating sequence numbers to each operation. ReBFT [30] is an approach that relies on a passive-replication paradigm to minimize the number of non-faulty replicas that participate in system operations in the absence of faults.

3 System and Threat Model

3.1 System Context

RepuCoin makes use of two types of blocks, namely keyblock and microblock. Keyblocks are created through PoW, as a means to elect a leader, and they do not contain any transaction. Microblocks are proposed by a randomly selected leader to record transactions into the blockchain.

All participants of the system, no matter whether they are miners or mobile clients, learn about new transactions and blocks in the same way as in BitCoin and other blockchains, through a peer-to-peer protocol.

In RepuCoin, we verify and commit microblocks, as well as decide on the keyblocks to be added to the blockchain, using Byzantine fault tolerance protocols (e.g., [28], [30]) with minor modifications, as presented in Section 4.4. Such form of agreement prevents a malicious leader from double spending a

coin, and resolves potential forks resulting from simultaneously mined keyblocks. According to Byzantine quorums theory [31], in order to reach an agreement, classic BFT protocols require votes from at least $2f + 1$ nodes,[1] to prevent an adversary controlling f nodes from breaking the protocol. *In practice, however, an open BFT-based system cannot guarantee that an attacker will never be able to control more than f nodes.* To enforce the assumption that no more than f Byzantine nodes are ever involved in a consensus, we introduce novel mechanisms to make it infeasible, in practice, for an attacker to seize f nodes within the consensus group, as detailed below.

3.2 System Model

RepuCoin is a system composed of a non-predetermined number of nodes, called *miners*. Each miner has a reputation score, which determines that miner's ability of obtaining rewards. A miner's reputation score is based on the correctness of its behavior and its regularity in adding blocks to the existing chain, hence correlated with the miner's computing power. RepuCoin considers a network that is untrustworthy and unreliable. In addition, we assume the network has partial synchrony [32].

To address the above-mentioned uncertainty (Section 3.1) in the definition of the consensus quorums and outcomes for open BFT-based protocols, RepuCoin resorts to two main techniques. First, we rely on the notion of having a *consensus group*, i.e., a subset of the miners denoted by $\mathbb{X}$ capable of controlling the operations of RepuCoin, namely running the consensus protocol. Second, the voting rules in the underlying consensus schemes are not nominal, but based on a novel *reputation-based weighted voting*. To this end, we refine the definition of quorums as follows: reaching agreement not only requires votes from $2f + 1$ nodes, but also demands that these nodes collectively have more than $\frac{2}{3}$ of the cumulative reputation of the consensus group.

The consensus group size is defined as the minimum number of miners with enough decision power (i.e., cumulative reputation) to ensure safe and live control of the system, given our quorum definition. Therefore, the consensus group members are obviously the miners with the highest reputation scores. This stratagem has two virtuous effects. First, RepuCoin's safety is guaranteed by consensus, which can be viewed as a deterministic control orchestrated by a set of miners with overwhelming cumulative reputation. By definition, such reputation itself gives an expectation of the correct behavior of these miners. Second, openess and fairness of RepuCoin relies on $\mathbb{X}$ being parametric and agnostic of identity of network members, as we show in Section 4.2. At configuration time, the size of $\mathbb{X}$ is calculated by meeting a target percentage of the overall decision power, and so it can be large or small, depending on the mining pool composition, but not pre-determined. On the other hand, as miners gain or lose reputation, they can (by merit or demerit) enter and leave the consensus group.

3.3 Threat Model

We consider a malicious (a.k.a. Byzantine) adversary, who can arbitrarily delay, drop, re-order, insert, or modify messages. We also consider collusions of an arbitrary number of miners, to model a malicious real organization capable of deploying a significant number of virtual miners under its direct dependence. We assume the security of the used cryptographic primitives, including a secure hash function and a secure signature scheme.

Such adversary can potentially control as many miners as it wishes, and coordinate them in real time with no delay. In consequence, the consensus group can be infiltrated by adversaries . However, we assume that the adversary has the ability to control at most $f \leq \lfloor \frac{|\mathbb{X}|-1}{3} \rfloor$ group members whose collective reputation is less than $\frac{1}{3}$ of the cumulative reputation of the members of consensus group $\mathbb{X}$. Under this assumption, the system is safe and live.

The coverage of this assumption, i.e., how to constrain the adversary's ability of infiltrating the consensus group, to meet the aforementioned assumption, is explained in Section 4.3. The dynamics of the reputation mechanism allow the consensus group controlling RepuCoin to be infiltrated with safety, making it practically infeasible for an attacker to break the system (as shown in Table 2 of Section 6).

4 RepuCoin

In this section, we present details describing the different concepts and modules underlying RepuCoin. In particular, Section 4.1 details the different types of blocks, the leader election mechanism, and the reward system proposed. Then, we present our reputation-based weighted voting mechanism in Section 4.2, and the reputation system in Section 4.3.

4.1 Block Mining and Reward System

As mentioned earlier, in order to support higher throughput rates, RepuCoin decouples leader election (keyblocks) from transaction serialization (microblocks).

Keyblock and Leader Election. Miners solve Bitcoin-like puzzles to create keyblocks, and receive rewards corresponding to keyblock creations. However, the keyblock creator is not necessarily the leader that commits transactions into microblocks. Rather, the leader is randomly elected from the reputable miners. The puzzle is defined as follows:

$$H(prev\_KB\_hash||Nonce||PK) < target,$$

where $H(\cdot)$ is a cryptographically secure hash function, $prev\_KB\_hash$ is the hash value of the previous keyblock, PK is the miner's public key , which the miner's reputation score R is associated with, and $target$ is a target value defined by the system. (For simplicity, we use reputation score and reputation interchangeably in this paper.)

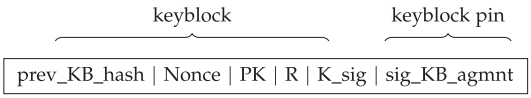
RepuCoin solves forked chains on the fly by dynamically forming the consensus group and agreeing on which chain to choose. The consensus group members are the top reputed miners in the mining network. The reputation score of miners can be calculated by using data from the blockchain, and is maintained locally by each miner. When different miners have the same reputation, a naive solution would be to order them according to their public key PK (a.k.a. address). However, this gives a miner with small PK an advantage. To avoid this, in RepuCoin, miners with the same reputation score are ordered by $H(PK, R)$, where reputation R (and

1. Hybrid BFT protocols with trusted hardware components, such as MinBFT [29], only require votes from $f + 1$ nodes.

therefore the hash value) is updated each time a new keyblock becomes part of the blockchain.

Each time a new keyblock is created, the creator proposes it to the consensus group. The group verifies the received keyblocks, and runs the underlying Byzantine agreement protocol to decide which keyblock to choose (if multiple conflicting keyblocks are proposed). We call a keyblock that is agreed upon and signed by the group a *pinned* keyblock. A pinned keyblock is final and canonical, it defines the unique global blockchain from the genesis block up to the pinned keyblock. All keyblocks that conflict with a pinned keyblock are considered invalid. New keyblocks are mined based on the hash value of the previous pinned keyblock.

The format of a pinned keyblock is as follows:



The (new) KB_hash is the hash of a pinned keyblock, i.e., all the material in the frame above, where K_sig is a signature on the hash value of $(prev\_KB\_hash, Nonce, PK, R)$, and sig_KB_agmnt is the signed agreement from the consensus protocol on committing this keyblock. The first keyblock is called the genesis block (as in Bitcoin), which is defined as part of the system. Note that to verify a keyblock, consensus group members check the validity of K_sig, the solution to the mining puzzle, and the reputation R.

Each time a new keyblock is pinned, the next leader—which verifies transactions and commits them into microblocks—is selected as follows:

$$l_i := x_j \quad s.t.\ x_j \in \mathbb{X} \ \wedge\ j = H(K\_sig_i) \bmod |\mathbb{X}|,$$

where l_i is the ith leader determined by the hash value $H(K\_sig_i)$ of the signature K_sig_i contained in the ith pinned keyblock, and $\mathbb{X}$ is the set of miners constituting the consensus group. Since a cryptographically secure hash function is considered a random oracle, the leader is selected randomly in the consensus group with probability $\frac{1}{|\mathbb{X}|}$. However, one concern with this leader selection process is that consensus group members can determine the following leader before pinning a block. Thus, a consensus member interested in getting more rewards would only accept (decide to pin) a block that makes itself the new leader. To address this issue, a simple approach is to determine a leader by using $H(sig_i)$ instead of $H(K\_sig_i)$, where sig_i is a signature on the current length of the blockchain issued by the keyblock creator. Note that it is important to issue this signature only after the keyblock has been pinned by the consensus group. This way, each consensus member would accept a block with equal probability.

Remark 1. Note that in consensus schemes, a sufficient number of signed votes determines an agreement, and only one valid agreement can be reached. However, different combinations of (a sufficient number of) signed votes can be used to form this agreement. So, even though the agreement is unique, the collection (i.e., sig_KB_agmnt) of signed votes may have different valid values. This results in different valid $prev\_KB\_hash$ on the same keyblock.

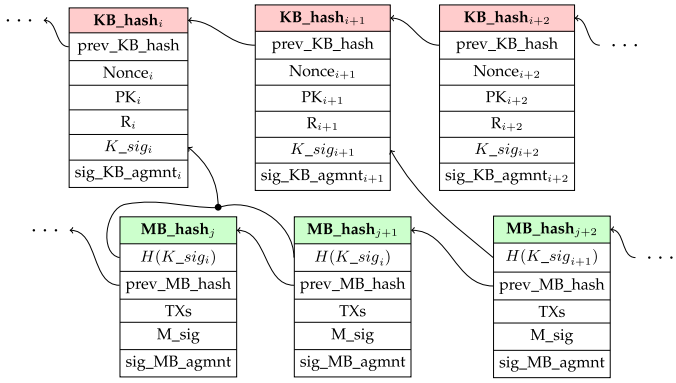
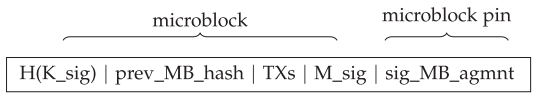


Fig. 1. A figure presentation of the blockchain structure.

Hence, miners may solve their puzzles based on different valid $prev\_KB\_hash$. However, this will not form a fork in the system, as only one of the valid solutions to the puzzle will be validated by the consensus group, as detailed in Section 4.2. In addition, upon reaching the agreement on the next keyblock, the $prev\_KB\_hash$ is considered the only valid hash value of the previous pinned keyblock, and only this value will be recorded in the blockchain.

Microblock. The current leader commits transactions into microblocks. To prevent double spending, each microblock is proposed to the consensus group before being accepted, and hence committing the transactions it encompasses. The group members verify the microblock, and initiate a consensus instance to agree on that microblock, which upon agreement is called a *pinned* microblock.

The format of a pinned microblock is shown below:



where $H(K\_sig)$ is the hash value of the K_sig contained in the current pinned keyblock, $prev\_MB\_hash$ is the hash value of the previous pinned microblock; TXs is a set of transactions organized as a Merkle tree, M_sig is a signature on the hash value of $(KB\_hash, prev\_MB\_hash, TXs)$, and sig_MB_agmnt is the signed agreement from the consensus protocol on the microblock. In this sense, in order to verify a microblock, consensus group members check the validity of M_sig, verify the hash values of the keyblock and the previous microblock, and verify the set of transactions TXs. If invalid transactions are detected, then the leader is punished, as presented in Section 4.3. The (new) MB_hash is the hash of the microblock (without a pin). In other words, the 'microblock pin' presented in the above frame is not part of the hash function's input. In this way, a leader can issue microblocks without waiting for the agreement, which optimizes the throughput of RepuCoin. The blockchain structure containing both keyblocks and microblocks is presented in Fig. 1.

Reward System. In RepuCoin there are two types of rewards, namely mining rewards and transaction fees. Upon successfully mining a keyblock, a miner is entitled to get a reward, precisely if that miner's keyblock gets pinned. This mining reward is of a pre-set amount.

Every transaction within the microblock carries a transaction fee. The randomly elected leader shares the transaction fees with the miner of the pinned keyblock according to Algorithm 1. Roughly speaking, the miner's reputation determines the number of microblocks from which it can obtain transaction fees; the leader gets the rest. However, a leader that can determine the microblocks from which it gets transaction fees, may optimize its income by putting transactions with higher transaction fees into these microblocks. To avoid this unfair game, RepuCoin uses the hash value of the next pinned keyblock, i.e., the keyblock pinned at the end of the new epoch, to decide which pinned microblocks are allocated to the miner and which go to the leader. Since the hash value of the next pinned keyblock cannot be predicted, RepuCoin eliminates the above situation.

More precisely, let $\mathbb{M} = \{m_0, m_1, \ldots, m_{n-1}\}$ be the sequence of n microblocks that are pinned by the consensus group. Let $R \in [0,1]$ be the reputation score of the miner which creates the $(i-1)$th pinned keyblock. The transaction fees contained in the set $\mathbb{M}'$ and $\mathbb{M}''$ of microblocks are shared between the miner of the $(i-1)$th pinned keyblock and the leader, respectively, as shown in Algorithm 1.

Algorithm 1. Reward Sharing Algorithm

Input: The sequence $\mathbb{M} = \{m_0, m_1, \ldots, m_{n-1}\}$ of microblocks pinned in the $(i-1)$th epoch, the signature K_sig_i contained in the ith pinned keyblock, and the reputation R of the miner who created the $(i-1)$th keyblock.
Output: Two subsets $\mathbb{M}', \mathbb{M}'' \subseteq \mathbb{M}$ of microblocks, where transaction fees contained in $\mathbb{M}'$ (resp. $\mathbb{M}''$) are allocated to the miner (resp. the leader) as reward.

1: $i' = H(K\_sig_i) \bmod n$
2: $k = 0$
3: $\mathbb{M}' = \emptyset$
4: **while** $k < R \cdot n$ **do**
5: $\quad j = i' + k \bmod n$
6: $\quad \mathbb{M}' = \mathbb{M}' \cup \{m_j\}$
7: $\quad k = k + 1$
8: **end while**
9: $\mathbb{M}'' = \mathbb{M} \setminus \mathbb{M}'$

The way transaction fees are shared between keyblock creators and leaders motivates miners to increase their reputation. First, keyblock creators with higher reputation gain higher shares of rewards. Second, highly reputed miners constitute the consensus group; hence they can become leaders and get shares of transaction fees.

Similar to the Bitcoin system, to spend a reward, the miner simply makes transactions by using the SK which is associated with the PK contained in the keyblock, and provides the hash of the keyblock or microblock as an input of the transaction.

In Bitcoin, a miner needs to wait a maturity period of 100 blocks to avoid non-mergeable transactions from forks. In RepuCoin, each pinned keyblock and its underlying pinned microblocks are canonical, so leaders do not need to wait for this period to avoid non-mergeable transactions.

Remark 2. With RepuCoin, miners, even the newly joined ones with initial reputation, will get much more reward than what they can get in Bitcoin. In particular, a miner with RepuCoin gains the same mining reward as with Bitcoin, and gains transaction fees that are at least 60 times as high as with Bitcoin. A detailed analysis can be found in our full report [20].

4.2 Block Pinning

In RepuCoin, we use consensus to pin both keyblocks and microblocks. Transactions belonging to pinned microblocks cannot be unrolled at a later point in time. In this section, we describe in more detail the underlying consensus mechanism we use to pin such blocks.

Byzantine fault-tolerant consensus algorithms typically rely on processes voting. A process needs to collect a quorum of votes on a given value/action for that value/action to be considered legal by the system. The size of the quorum is selected in a way that guarantees (i) safety of decisions, e.g., avoiding conflicting decisions, and (ii) liveness of the system, i.e., miners should be able to hear eventually a number of votes on some value/action from a quorum of miners. It has been shown that in systems, as soon as $\frac{1}{3}$ or more of the miners are compromised, an attacker can make the system inconsistent—an attacker can make different parts of the system decide differently [31]. In order to make our system robust to such attacks, we propose to modify the traditional nominal voting mechanism, i.e., hearing from a sufficient number (quorum) of miners, by requiring as well to hear from a sufficient number of miners such that their added reputation is above a defined threshold. Such a modification prevents an attacker from breaking the correctness of the system directly upon compromising any $\frac{1}{3}$ of the miners: it should compromise as well enough miners that their added reputation is at least $\frac{1}{3}$ of the total reputation of the consensus group. Details on how to adapt a chosen BFT protocol to RepuCoin are presented in Section 4.4.

Consensus in RepuCoin employs a novel *reputation-based weighted voting mechanism*, i.e., rather than treating each vote from consensus group members equally (e.g., as in classic Byzantine protocols), the weight of each vote becomes its reputation over the total reputation of the group. More precisely, let $\{x_1, \ldots, x_{|\mathbb{X}|}\}$ be the consensus group, and each member x_i of the group has its reputation score R_i. The weight of x_i's vote is $\frac{R_i}{\sum_{i=1}^{|\mathbb{X}|} R_i}$, for all possible x_i.

Instead of only waiting to hear from at least $2f+1$ nominal members to validate a value or an action, it is also necessary that the collective reputation of those members is more than $\frac{2}{3}$ of the total reputation of the consensus group.

Remark 3. Weighted-voting [33] is a classic and well known concept. The novelty of our weighted-voting system comes from the way that this weight is defined. More precisely, the weight of a miner's vote is given by this miner's reputation, i.e., its 'integrated power'. In particular, it considers the quantity and regularity of contributions over the entire blockchain, and provides a model to punish misbehaved miners (see section 4.3). In other words, we constrain the evolution of reputation over time. Thus, unlike in traditional systems, it becomes significantly difficult for an attacker to re-obtain enough voting power after a deviation, or to flash-build it like in flash attacks.

Consensus Group. As mentioned in Section 3.2, the size $|\mathbb{X}|$ of the consensus group is not pre-determined, but rather calculated by meeting a target percentage[2] of the overall decision power. We select the members of the consensus group based on our reputation system; namely, the $|\mathbb{X}|$ miners with the top reputations constitute the members.

We define an epoch as the period between any two successive keyblocks that become part of the blockchain. Every epoch possesses a leader, which is the miner that should issue a maximum pre-specified amount of microblocks. In every epoch, the reputation of only one miner, the creator of that pinned keyblock, may gain an extra increase; the reputation of all other miners would only have a very minor change according to Algorithm 2, or drops to "0" if they lie (see Section 4.3). Accordingly, given that $f \leq \lfloor \frac{|\mathbb{X}|-1}{3} \rfloor$ can be malicious, the members of the consensus group in any two consecutive epochs can differ by at most f members. This stability in the members of the consensus group of consecutive epochs ensures the safety of consensus decisions. Namely, at the beginning of a new epoch correct consensus group members are aware of all committed transactions and hence do not accept/validate any conflicting transactions proposed by the new leader.

Committing Microblocks. The leader of the current epoch, issues transactions in the form of microblocks. After generating a microblock, the leader initiates a consensus instance for this microblock proposing an accept to commit that micro-block. Other consensus members will propose to either accept or decline (by not accepting) this micro-block, depending on the transactions contained within and of course their validity. In other words, if transactions within the micro-block are invalid then members should decline committing that microblock. The leader continues to issue microblocks that are proposed to the consensus group for validation and commitment, until a new leader is elected.

Committing Keyblocks. Upon successfully mining a keyblock, the miner of that block sends that keyblock to all members of the consensus group.

Upon receiving a keyblock, this group member initiates a consensus instance proposing the received keyblock (first received keyblock in the case when many such keyblocks are received). As a result, the members of the consensus group decide on a single keyblock to be part of the blockchain. The miner of that block is termed as the "winner". The hash of the new keyblock output by the consensus decides which member of the consensus group becomes the leader of the current epoch as previously mentioned.

A member of the consensus group that successfully decides on the identity of the new leader stops validating microblocks relative to the previous leader. Afterwards, that member initiates a consensus instance to agree on the total set of committed microblocks. Consensus group members need to agree on the total set of microblocks, since a leader is selected from this group. A leader that does not know the total set of committed microblocks might propose microblocks that are in conflict with committed ones and accordingly lose its reputation, not out of maliciousness but simply out of lack of knowledge. To avoid this situation, every member after reaching a decision on the identity of the new leader submits to consensus the largest sequence number of microblocks that have been committed along with a verifiable proof of this claim. Namely we assume that all consensus algorithms we use are implemented using digital signatures. As such, having a sufficient number N of signatures on a decision constitute a proof of its validity. We say the number N of signatures is sufficient if $N \geq 2f + 1$ and if the total reputation of the miners issuing these N signatures is more than $\frac{2}{3}$ of the total reputation of the group.

Upon reaching a decision on the new leader and on the global set of committed microblocks, each consensus group member also sends a message to notify the winner, the current leader, and the newly elected leader of this result. A consensus group member waits to either hear from consensus or from a sufficient number N' of other group members about the identity of the new leader and the global set of committed microblocks, before it adopts that member as leader. We say N' is sufficient if the total reputation of the issuers of the N' signatures is more than $\frac{1}{3}$ of the total reputation of the group and if $N' \geq f + 1$. In that case, the current leader simply stops issuing microblocks and the new leader takes over proposing microblocks.

Optimizing Agreement on Committed Microblocks. In order to have agreement on the set of committed microblocks without resorting to a consensus instance, we propose the following optimization. Due to the PoW, it is known that mining a keyblock successfully takes a certain time depending on the mining difficulty (e.g., on average 10 minutes in Bitcoin). If we assume that a leader issues microblocks to be committed at a pre-specified rate, then we can assume that on average a leader commits m microblocks per epoch. Accordingly, all consensus group members do not validate or commit more than m microblocks for any given leader. Upon reaching a decision on the identity of a new leader (as a result of having a new keyblock mined) a consensus group member only initiates consensus on the set of committed microblocks if it has not seen m committed microblocks from the previous leader.

The benefits of fixing the number of microblocks (that a leader can commit) to m microblocks per epoch extends beyond having a fast and efficient agreement on the set of committed microblocks. It can also be used to incentivise leaders not to hinder throughput, e.g., a malicious leader in the worst case might decide not to submit any microblocks to intentionally stall the throughput. However now, since a leader is expected to commit m microblocks per epoch, leaders which cannot meet that constraint can be punished for example by decreasing their reputation and hence decreasing their chances of staying part of the consensus group and becoming leaders again.

4.3 Reputation System

This section describes our reputation system and the proof-of-reputation. We first highlight the shortcomings of previous systems. For example, a proof-of-work based system requires a miner to show that it has done some work in order to include its set of proposed transactions, and hence extend the chain. Thus, a miner that has a high computing power can join the system at any time and can play attacks. Similarly, in

2. Similar to the parameters of other systems, such as the block size of Bitcoin or the window size of ByzCoin, the target percentage of the overall decision power is a system parameter that can be reconfigured if necessary.

TABLE 1
The Notations

Notation	Explanation
L	the length of the current blockchain;
c	the size of a block chunk, pre-defined by the system;
t	$t = \lceil \frac{L}{c} \rceil$ is the number of block chunks contained in a blockchain with length L;
Ext	the optional external source of reputation for the miner;
H	a binary presenting whether the miner is honest ("1") or not ("0");
k_i	the number of keyblocks created by the miner in chunk i;
N_l	the number of times that the miner is elected as a leader;
m_j	the number of valid microblocks created by the miner at the jth time it is the leader;
m	the maximum number of microblocks that a leader is allowed to create, as defined by the system.
mean_i	the mean value of keyblocks (if $i = k$) or microblocks (if $i = m$) created by a miner or a leader across all epochs in the blockchain, respectively.
s_i	the standard deviation corresponding to mean_i, for $i \in \{k, m\}$.
(a, λ)	reputation system parameters

the proof-of-membership system [15], a miner has to show that it has created enough blocks recently to demonstrate its computing power, then it can issue microblocks and can gain power in the consensus protocol. Again, an attacker with higher computing power can join the system at any time and can break the system. However, with proof-of-reputation, in addition to creating enough recent keyblocks, a miner has to show that it has behaved honestly and created keyblocks regularly for a period of time before being able to launch any attacks on the system.

Given the blockchain, the reputation of any miner can be calculated at any point in time. Accordingly, each miner maintains its own copy of the reputation score of all miners, based on the globally agreed blockchain. We denote by R the reputation of a miner, which can take values in $[0, 1]$. R is calculated according to Algorithm 2. The notations are defined in Table 1.

In particular, $Ext \in [0, 1]$ is the (optional) external source reputation of the miner. For example, when Citybank joins RepuCoin, it may have a starting reputation that is higher than a random individual joiner. In RepuCoin, this is encoded by using Ext.[3] $H \in \{0, 1\}$ is the honesty of the miner, which is set to "1" for each new joiner, and is set to "0" if a miner has misbehaved.[4] A miner is said to misbehave if:

- it presents conflicting signed messages to other consensus group members; or
- it commits microblocks with conflicting transactions when the miner is elected as leader.

3. Note that this is only used to optimize the reputation system. However, to study the worst case, our analysis in Section 6.3 also shows the security guarantee without having this external source of reputation.

4. Once the honesty H of a miner has been set to "0", the Ext of the corresponding entity will be set to "0" as this entity is not trustworthy.

Algorithm 2. Reputation Algorithm

Input: L, $\{k_i\}_{i=1}^{t}$, $\{m_j\}_{j=1}^{N_l}$, m, c, a, λ, H, and Ext.
Output: Reputation $R \in [0, 1]$ of the corresponding miner.

$\text{mean}_k = \frac{\sum_{i=1}^{t} k_i}{L}$
$\text{mean}_m = \frac{1}{N_l} \cdot \sum_{j=1}^{N_l} \frac{m_j}{m}$
$s_k = \sqrt{\frac{1}{t} \cdot \sum_{i=1}^{t} (\frac{k_i}{c} - \frac{\sum_{i=1}^{t} k_i}{L})^2}$
$s_m = \sqrt{\frac{1}{N_l} \cdot \sum_{j=1}^{N_l} (\frac{m_j}{m} - \frac{1}{N_l} \cdot \sum_{j=1}^{N_l} \frac{m_j}{m})^2}$
$y_1 = \frac{\text{mean}_k}{1+s_k}$
if $N_l \geq 1$ **then**
    $y_2 = \frac{\text{mean}_m}{1+s_m}$
else
    $y_2 = 1$
end if
$x = y_1 \cdot y_2 \cdot L$
$f(x) = \frac{1}{2}(1 + \frac{x-a}{\lambda+|x-a|})$
$R = min(1, H \cdot (Ext + f(x)))$

Upon their occurrence, an evidence of such misbehavior is included in the blockchain as a special transaction, similar to past work [13], [34]. Non-Byzantine miners are incentivised to place such a proof of fraud into the blockchain, to make malicious acts visible to everyone, hence preserving the health of the system. If a cryptocurrency system is not healthy, then its users will lose their confidence in the system. This may result in the plummeting of its currency exchange rate, and all miners will have a loss. So, miners are incentivised to keep the health of the system for their own profit.

The Reputation Function. We intended to define the social objectives of reputation in RepuCoin in a precise and parameterizable way. Those objectives are: (i) careful start, through an initial slow increase; (ii) potential for quick reward of mature participants, through fast increase in mid-life; (iii) prevention of over-control, by slow increase near the top.

The formula defining the progression (resp. regression) of reputation, $f(x)$ above, is a sigmoid function. It ensures that miners, at the start, can only increase their reputation slowly, even if having a strong computing power. A miner needs to stay in the system and behave honestly for a long enough period, to progressively increase its reputation up to the turning point, where it is trusted enough to be incentivized to make it grow more quickly, to more interesting levels. And finally, the curve inflects again, so that the reputation does not grow forever, but asymptotically reaches a plateau that promotes a balance of power amongst miners. The reputation function is also parameterized, to allow to mark these points precisely, namely the parameters (a, λ) can be tuned to adopt changes on when and how fast/slow miners can increase their reputation. The slope of $f(x)$ is directly correlated with the value of λ. The inflection point of $f(x)$ occurs at $x = a$, and is the point where a miner's reputation growth rate starts to decline.

We denote by a block chunk (or just 'chunk' for simplicity) a sequence of successive keyblocks in the blockchain. Blocks chunks satisfy the following: (i) all block chunks are of the same size, and (ii) any keyblock is included in exactly one block chunk.

y_1, defined at line 5 of Algorithm 2, captures the miner's "regularity" of generating keyblocks in each block chunk. In other words, y_1 *shows how regularly the miner contributes its computing power to the system*.

In particular, the numerator mean_k of y_1 is the percentage of pinned keyblocks generated by the miner, represents the fraction of valid work that a miner has contributed to the whole system. In the denominator, s_k is the standard deviation of the pinned keyblocks generated by the miner, indicates the regularity with which a miner contributes to every chunk. Together, they guarantee that a miner's reputation is computed based on the miner's *integrated power*. Hence, a miner's integrated power is given by the total amount of valid work a miner has done over the period of time it has been active and the regularity of that work in the entire blockchain, rather than the miner's mining ability at a given time (or instantaneous power) as in classic proof-of-work. As such, when the system has been operated for some time, even a miner with strong computing power cannot build-up its reputation quickly: it needs to contribute honestly and regularly to the system to gain reputation. We present a more detailed analysis in Section 6.3.

Similarly, y_2 represents the "regularity" with which a leader commits the defined number of microblocks when it is selected. This incentivises leaders to optimize the throughput of RepuCoin.

4.4 Adapting Existing BFT Protocols

RepuCoin uses existing leader-based BFT protocols supporting digital signatures, to pin blocks. Apart from modifying the weight of the votes, RepuCoin also requires the following changes to adapt the existing BFT protocols.

Secure Bootstrapping. With the potentially unbalanced amount of mining power in different blockchains, how to do secure bootstrapping is an open challenge in all blockchain systems. This, however, is not a problem with the classical BFT protocols, where the set of participants are predefined and fixed. Thus, to adapt existing BFT protocols, we need to provide a mechanism to establish a secure way to initialize consensus group, when no keyblock is created.

In RepuCoin, we assume the existence of a social community where participants vote to make decisions on several aspects of the system, such as the security parameters and external reputation factors. Such community exists for almost all permisionless blockchains. For example, with BitCoin this is the community who votes for proposals such as changing the maximum block size.

This community in RepuCoin votes a set of parties with external reputation to ensure a controlled bootstrapping, and record the result in the genesis block. The parties with initial higher external reputation will form the consensus group. During the secure bootstrapping phase, other miners with small or zero external reputation will gradually gain reputation and enter the consensus group.

View Change. Classical BFT protocols provide view change—leader election and membership update—as a housekeeping function in the course of failures or recoveries in the (static) system participants roster. In addition to these technical functions, these protocols can be used in non-standard ways in blockchain consensus. That was the case for example of ByzCoin, where a new leader is elected every time a new keyblock is created, by invoking the PBFT view-change protocol [15]. Similarly, with RepuCoin a view change is enforced by the pinning of a new keyblock, which ends an epoch, and thus establishes a consistent cut where the systems flushes (achieving consensus on the blockchain closing the epoch). Thus, several operations can be safely performed at this clean (re-)starting point: (i) a new consensus leader is elected; (ii) and the consensus membership is updated.

To adapt existing classical PBFT-like systems, the first difference is that, for (i), the leader election is deliberately provoked in this case (not on account of e.g., a failure) and the criterion changes to random selection, as presented in Section 4.1. For (ii), what happens is, again, a non-standard redefinition of the membership: the (ending epoch) consensus group re-evaluates the rule for consensus group formation (a quorum of the top reputed miners, see Section 4.2). Note that this can directly and deterministically be derived from the data in the blockchain, so consensus is safely achieved on the new roster of $\mathbb{X}$, which is installed for the new epoch. We recall that these operations occur through stable and safe states of the system, as mentioned before.

Crash/leave Detection. As this is a permissionless environment, any miner can join, leave or crash at any time. If a consensus group member left the system, then RepuCoin will eventually detect that this has happened, by checking whether this member has been involved in the last instances of the consensus. If it is the epoch leader, view change ensues in the usual manner in BFT consensus protocols.

Message Size. Most existing BFT protocols have been designed for state machine replication, and even optimized for short messages/commands. Performance shown in existing publications mostly concerns tests with relatively "small" block sizes. As shown in [35], the impact of largely increased block sizes on blockchain consensus performance should not be neglected either when choosing existing protocols, or when designing blockchain-specific BFT protocols.

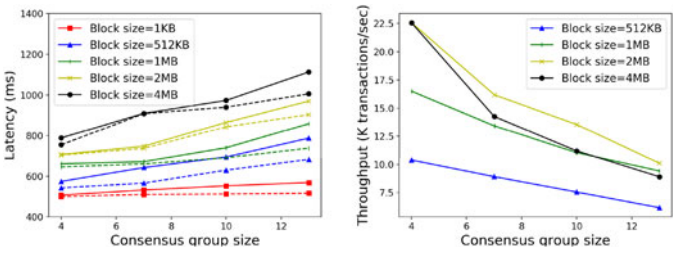
5 Performance Evaluation

5.1 Implementation

Setup. We extend the BFT-SMaRt [36] library for our RepuCoin implementation. We deploy each member of the consensus group on a different machine, each having the following specifications: Dell FC430, Intel Xeon E5-2680 v3 @2.5 GHz, 48 GB RAM. To simulate wide-area network conditions, we impose a round-trip network latency of 200 ms between any two machines, and a maximum communication bandwidth between any pair of machines to 35 Mbps. To better simulate the system in the real world scenario, *we make use of the mining power distribution from the Bitcoin mining network*.[5] More details and justifications of our setting, including the choice of PoW mining rate and mining power distribution, can be found in our technological report [20].

Consensus Goup. We consider consensus groups that initially control from about 50 to 98.1 percent of computing power. With the current Bitcoin mining power distribution,

5. The sequence of computing power of the top 24 pools is (15.1, 10.1, 10.0, 9.5, 8.3, 7.1, 6.4, 5.9, 5.5, 4.0, 2.9, 2.8, 2.4, 2.2, 1.7, 1.5, 1.5, 0.7, 0.5, 0.5, 0.3, 0.3, 0.2, 0.2), respectively. https://bitcoinchain.com/pools (as of April 2017)



(a) Consensus latency (b) Throughput.

Fig. 2. Performance evaluation, where (a) provides a comparison of consensus latency between BFT-SMaRt (dashed lines) and RepuCoin (straight lines); and (b) shows the throughput of RepuCoin.

the corresponding consensus group sizes would range from 4 to 19. However, since we show, in Section 6.3, that security is hampered for consensus groups that control more than 90 percent computing power, given Bitcoin's computing power distribution, we only present the performance results with consensus group controlling computing power from 44.7 to 90 percent.

5.2 Consensus Latency

In this section, we measure the latency of our consensus implementation , and compare it with the latency of the original BFT-SMaRt. Such a comparison illustrates the timing overhead that is incurred relative to using our reputation-based weighted voting mechanism. We recall that in order to reach consensus, BFT-SMaRt requires at least $2f + 1$ members to agree on a value, while our reputation-based weighted voting variant requires in addition that these members (which are at least $2f + 1$) have collectively more than $\frac{2}{3}$ of the reputation of the entire consensus group.

We run experiments using keyblocks of size 1 KB and microblocks of sizes 512 KB, 1 MB, 2 MB, and 4 MB. Unlike microblocks, keyblocks are typically small in size as they do not contain any transactions. Fig. 2a presents the consensus latency of RepuCoin. It shows that RepuCoin and BFT-SMaRt have a similar consensus latency values and patterns. For example, in both RepuCoin's consensus and BFT-SMaRt, consensus latency increases dramatically with the block size. The reason behind this trend can be explained by two things. First, it takes longer for the leader to propose microblocks to the consensus group, and for the group members to transmit a batch of the PROPOSE message which contains un-hashed microblock. Second, computing the hash value of a larger block and verifying the transactions it contains consume more time.

Moreover, in both RepuCoin's consensus and BFT-SMaRt, when the group size increases from 4 (which controls 44.7 percent computing power of the network) to 13 (which controls 90 percent computing power of the network), the consensus latency increases by more than 50 percent. However, despite this increase in latency, consensus can be reached in about 0.5-1.2 second, even when considering the blocks of size 4 MB.

5.3 Throughput

Fig. 2b presents the throughput of RepuCoin. First, as expected, our results in Fig. 2b show that the smaller the consensus group the higher the throughput. For example, using 2 MB microblocks, the throughput increases from slightly more than 10000 TPS with a consensus group of size 13 (controlling 90 percent computing power), to 22500 TPS with a consensus group of size 4 (controlling 44.7 percent computing power). Second, for all group sizes, one can see, as expected, that the throughput tends to increase as blocks become larger, and this is what we observe up to 2 MB. For example, when the consensus controls 90 percent computing power of the entire network (group size of 13), the throughput for blocks of 512 KB, 1 MB, and 2 MB, is respectively equal to 6200, 9400, and 10000 TPS. We observe that using larger block sizes (e.g., 4 MB), decreases the throughput. However, this outlier is an artefact of the underlying protocol we use, i.e., the BFT-SMaRt library, whose sheer performance, as a regular BFT protocol, is seemingly affected for very large block sizes, as discussed in Section 4.4.

When RepuCoin provides the best security guarantee (when the consensus group controls 90 percent computing power, as shown in Section 6.3), RepuCoin achieves a throughput of 10000 TPS when using 2 MB blocks. This means that RepuCoin can handle the average transaction rates of Paypal and VISA as measured in real-life, which are 115 TPS and 1700 TPS respectively.

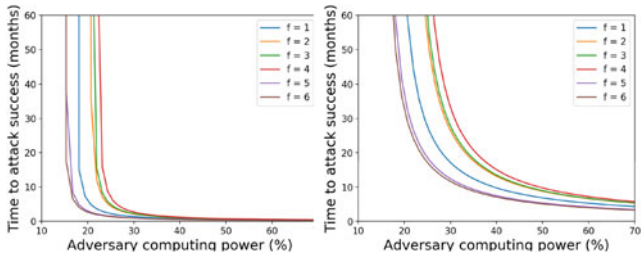
According to a survey [37], our throughput, i.e., 10K TPS, is outstanding among the analysed systems, as the reported peak figure for permissionless ledgers is also 10k TPS. We refer readers to the survey for more details.

6 Security Analysis

In this section, we present our analysis of the security provided by the mechanisms used in RepuCoin, namely reputation-based weighted voting consensus and proof-of-reputation function. We begin by discussing the safety and liveness correctness conditions of the reputation-based weighted voting consensus protocol in Section 6.1, with pre-defined bounds on the relative reputation scores of participants. Then, we present in Section 6.2 a theorem (proven in [20]) showing that the achievement of those scores, which give decision power, is physically bounded by the conditions imposed on the growth rate of the proof-of-reputation function. Next, in Section 6.3, we present experiments exemplifying concrete values for the decision power growth versus time in several situations, showing that RepuCoin indeed achieves very high stochastic robustness against attacks on its liveness or safety. Moreover, in Section 6.4 we provide an analysis of the non-rationality of infiltration attacks, with a comparison on the cost of attacking different systems. Finally, we describe in detail how RepuCoin prevents known attacks in Section 6.5.

6.1 Reputation-Based Consensus Safety and Liveness

Unlike PoW-based systems, when using proof-of-reputation, an attacker cannot break the system by merely relying on its mining ability, i.e., its computing power. An attacker rather needs to gain reputation and hence contribute to the blockchain, by yielding pinned keyblocks. We recall that the reputation of a miner with correct behavior, in RepuCoin, builds essentially on its continued and regular contribution to the entire blockchain in addition to its external source of reputation Ext.



(a) Attack after 3 months (b) Attack after 12 months

Fig. 3. Minimum effort to break the liveness of RepuCoin. We consider cases where an attacker joins after the system has operated for 3 months and a year, and where the consensus group controls mining power ranging from 44.7 percent (i.e., f = 1, $|\mathbb{X}| = 4$) to 98.1 percent (i.e., f = 6, $|\mathbb{X}| = 19$). The *x*-axis shows the needed computing power and the *y*-axis shows the required time to attack the system.

To study the worst case scenario, we do not consider any miner to have established an agreed upon trust, i.e., external reputation $Ext = 0$ for all miners. For presentation simplicity we assume that every miner behaves honestly for some period of time that allows the attacker to have a sufficient reputation when intending to attack the system.

Let $\mathbb{X} = \{x_1, \ldots, x_{|\mathbb{X}|}\}$ be the consensus group. We note R_i the reputation score of miner x_i, which gives it decision power. RepuCoin can rely on any underlying secure consensus algorithm, that can be adapted according to Section 4.4, to guarantee safety and liveness.

RepuCoin guarantees consensus *safety*, if: (i) the attacker controls no more than f miners in the consensus group; or (ii) the consensus group members compromised by the attacker have a total reputation R_A such that

$$R_A < \frac{\sum_{i=1}^{|\mathbb{X}|} R_i}{3},$$

where R_i is calculated according to Algorithm 2. In other words, an attacker cannot break the safety unless both (i) and (ii) do not hold.

In addition, if any of the above two conditions does not hold, then an attacker can break the *liveness* of the system, i.e., an agreement may not be made on any block.

6.2 Bounded Proof-of-Reputation Function Growth

We hook the stochastic equations governing the evolution of our system to 'physics': it remains impossible to gain power faster than some upper bound derived from the need to perform a number of continued honest contributions to the network—what we called 'integrated power', to differentiate from 'instantaneous power', haunting all previous works by leading to flash attacks.

Our Theorem 1 (proven in [20]) shows that the proof-of-reputation function growth rate is bounded. In consequence, the rate of change of the decision power mentioned in the previous section is, at any time, limited.

Theorem 1. *In RepuCoin, if the mining power of each participant remains the same, then at any time of the system, the rate of reputation increase of any node is bounded by* $\frac{1}{2\lambda}$*, and the corresponding increase of the decision power of any consensus group member* i *is bounded by* $\frac{1}{2\lambda}\Delta mean_{k,i}\Delta L$.

Thus, with RepuCoin, the rate of increase of decision power in the entire system is limited regardless of the newly joined computing power. This makes RepuCoin secure against flash attacks launched by a late joiner, even when the attacker has a large amount of computing power. We refer readers to [20] for the full analysis and proofs.

TABLE 2
The Minimum Computing Power (CP) and Cost Required to Break the Liveness of RepuCoin within a Targeted Time Period, When an Attacker Joins the System at Different Times

Joining time\ Target	1 week	1 month	3 months	6 months
1 month	infeasible	CP: 45%; BTC: *635; BYZ: *6	CP: 30%; BTC: *1271; BYZ: *11	CP: 27%; BTC: *2287; BYZ: *20
3 months	infeasible	CP: 90%; BTC: *1270; BYZ: *11	CP: 45%; BTC: *1906; BYZ: *17	CP: 33%; BTC: *2795; BYZ: *25
6 months	infeasible	infeasible	CP: 68%; BTC: *2880; BYZ: *26	CP: 45%; BTC: *3812; BYZ: *34
9 months	infeasible	infeasible	CP: 90%; BTC: *3812; BYZ: *34	CP: 54%; BTC: *4574; BYZ: *41
12 months	infeasible	infeasible	infeasible	CP: 68%; BTC: *5760; BYZ: *51
18 months	infeasible	infeasible	infeasible	CP: 91%; BTC: *7708; BYZ: *69
20 months	infeasible	infeasible	infeasible	infeasible

sys : $*N$ means that the cost of attacking RepuCoin is at least N times as high as the cost of attacking sys, where sys is either Bitcoin (BTC) or ByzCoin (BYZ).

6.3 Proof-of-Reputation Attack Resilience

So far, we have shown analytically that: RepuCoin is safe and live (Section 6.1) as long as decision power of attackers is below a defined threshold; then, we showed (Section 6.2) that it takes a known and bounded effort for attackers to reach that threshold and to control the consensus group. This section analyses how much would the attack effort be to reach the above-mentioned control.

Fig. 3 and Table 2 show the requirements both in terms of the computing power and the time that should be spent doing honest work in the system, in order for an attacker to successfully launch any attack. In other words, they present the minimum effort to attack the liveness of RepuCoin.

Fig. 3 indicates that the system is most secure when the consensus group $\mathbb{X}$ controls 90 percent computing power (with 13 nodes, i.e., $f = 4$), and is most vulnerable when it controls 98.1 percent computing power. In fact, we can observe that the system (when $\mathbb{X}$ controls less than 90 percent computing power) becomes more secure as the consensus group controls more computing power of the network. After that increase the group size begins to depreciate the system security. These results can be explained by the fact that when the consensus group size grows beyond some point, the distribution of the computing power and reputation in the enlarged group could highly vary. For example, when $\mathbb{X}$ controls about 100 percent computing power of the system, more miners with relatively low reputation might become part of the consensus group; hence an attacker needs less time (and reputation) to infiltrate the consensus group and launch attacks. Our results, in Fig. 3, show that if the consensus group controls 90y initial computing power, then an attacker

joining after 3 months of system operation with 26, 34, and 51 percent computing power of the entire network would need to work honestly for 22 months, 6 months, and 2.4 months respectively, to break the liveness of the system. If an attacker joins after 1 year, then it is infeasible for this attacker to break the system's liveness (and thus the system's safety) with a computing power $\leq 26\%$; and the attacker would need 2 years (resp. 10 months) when possessing 34 percent (resp. 51 percent) of the system's computing power. We say that it is infeasible for an attacker with $\leq 26\%$ to successfully launch attacks, as our analysis shows that the attacker would have to contribute to the system honestly for 108 years before being able to do so. It is worth noting that an attacker with computing power p_a joining a system whose computing power is p_s would have $\frac{p_a}{p_a+p_s} \times 100\%$ of the system's computing power.

In Table 2, we provide a different view on the attackers ability of successfully attacking the system's liveness. Breaking RepuCoin's safety is even harder than, at best as difficult as, breaking its liveness. It shows that an attacker who wants to break the system within one month after joining, i.e., by making the system lose liveness, would need to control at least 90 percent of the system's computing power, if that attacker joins after 3 months of the system being in operation. An attacker joining the system at a later time, e.g., 1 year (resp. 1.5 years) after the system operation, would never succeed in breaking the system's liveness nor safety within a period of 3 months after joining, and would require at least 68 percent (resp. 91 percent) of the system's computing power to launch at attack within 6 months after joining.

6.4 Non-Rationality of Infiltration Attacks

We hook heuristics to well-founded 'rationality': there is basically no rational economic model in RepuCoin that makes it worth to attack the network, i.e., it always costs much more than what would be gained, due to the following reasons.

First, attacks can only be successful after gaining enough reputation, by means of a lot of past investment and time spent (unlike all previous works, based on instantaneous power, a.k.a. computing power, which can be harnessed in several expeditious ways). Second, reputation goes to zero after the first detected attack (unlike all previous works, which essentially don't have memory and allow repeated attacks). Last but not least, the bribery attack by buying reputation is also made ineffective. More precisely, an adaptive adversary may try to acquire one or more nodes with high reputation, in order to trigger a 'flash reputation' attack. However, buying that reputation based power should cost at least as much as the investment previously made by the sellers, and the gain upon first use (and last, since reputation goes to zero), would never match the expense. So, it always costs much more than what would be gained.

To better illustrate the cost in different systems, Table 2 shows the cost of successfully attacking RepuCoin in comparison to the cost of attacking Bitcoin and ByzCoin, where the former is the most impactful system to date, and the latter is the only existing system where the voting power is also accumulated using mining power. For the analysis, we make use of the Bitcoin real-world mining power distribution, as presented in Section 5.1.

To successfully attack Bitcoin, in the best case (not considering the selfish mining attack), an attacker needs to have 51 percent of the computing power, and is required to maintain this computing power only for about an hour if 6 confirmations are required, to mine its own private chain on the side. Let α be the computing power (in unit) of the entire network, then the cost for each successful attack on Bitcoin is about $0.51\alpha r$, where r is the price of maintaining 1 unit of computing power per hour.

With ByzCoin, in the best case (not considering the selfish mining attack), an attacker needs to have 34 percent computing power, and to maintain this power for the entire window (i.e., 1008 blocks), which is about a week. Thus, the cost is about $168 \cdot 0.34\alpha r$, which is $57.12\alpha r$. The cost of repeating this attack is the same, i.e., $57.12\alpha r$.

With RepuCoin, *in the worst case*, where an attacker joins at the beginning of the system, RepuCoin does no better than Bitcoin and ByzCoin upon the first attack. However, to repeat the same attack, the attack would cost much more, as the reputation of the attacker would go to zero, and the attacker would be considered a late joiner.

For a later joined attacker, the minimum cost can also be calculated based on the required computing power, as shown in Table 2. For example, for a 6-month late joined attacker, to successfully attack the system within 3 months, the cost of attack is about $2160 \cdot 0.68\alpha r$, which is about $1469\alpha r$. That is, the cost is 26 times as high as the cost of attacking ByzCoin, and 2880 times as high as the cost of attacking Bitcoin. Taking another example scenario, for a 1-year late joined attacker, it is infeasible to successfully attack the system within 3 months, even with the computing power of the entire network. However, the attacker is able to attack the system within 6 months with a cost of $4320 \cdot 0.68\alpha r$, which is about $2938\alpha r$. In this case, the cost is 51 times as high as the cost of attacking ByzCoin, and 5760 times of the cost of attacking Bitcoin.

TABLE 3
Summary Comparison of Attack Resilience

Attacks	Bitcoin	Bitcoin-NG	ByzCoin	RepuCoin
Flash attack	×	×	×	√
Selfish mining attack	×	×	×	√
Attack on consistency	×	×	√	√
Attack on liveness	√	√	×	√
Double spending attacks	×	×	√	√
Eclipse attacks	×	×	√[1]	√[1]

√ – *The system is secure against this attack.*
× – *The system is vulnerable to this attack.*
[1]*The system is secure against eclipse attacks for double spending purpose, however, if an attacker is able to partition the network, then it can temporally delay the consensus process and reduce the throughput.*

6.5 Defense against Specific Attacks

This section discusses defences of existing protocols against known attacks. Table 3 summarizes a comparison between Bitcoin, Bitcoin-NG, ByzCoin, and RepuCoin. More detailed discussions can be found in our report [20].

Flash Attacks. In flash attacks [8], an attacker is able to obtain a temporary majority of computing power by renting enough mining capacity. This would break the security assumption of classic proof-of-work based systems.

RepuCoin, however, is resilient to flash attacks. As shown in Section 6, even an attacker with high computing power, depending on when that attacker joins, might require a very

long period of time before being able to gain enough reputation to harm the system.

Selfish Mining Attack. In a selfish mining attack [6], an attacker keeps its mined blocks private, and publishes them according to some strategy that would allow the attacker to claim all rewards (with $>25\%$ computing power). We refer readers to [6], [7], [16] for more details on the attack, its generalization and its optimization.

RepuCoin pins each created keyblock, and new keyblocks can only be created based on the pinned keyblock. Given that RepuCoin relies on a reputation-based consensus and a secure signature scheme, no attacker can predict the hash value of a pinned block without controlling at least $2f+1$ consensus group members that collectively have more than $\frac{2}{3}$ of the reputation of the entire consensus group. So, a selfish miner cannot gain any advantage in RepuCoin by hiding its created blocks.

Blockchain Consistency and System Liveness. Although Bitcoin-NG provides a high transaction throughput, it does not solve or address the consistency issues of Bitcoin. Namely, all transactions are only probabilistically valid. ByzCoin addresses these consistency issues providing deterministic transaction guarantees while achieving a high throughput. However, ByzCoin can permanently lose liveness during reconfiguration [17], [27], and a malicious miner can repeatedly make ByzCoin lose liveness temporarily [15].

RepuCoin provides strong transaction consistency and better liveness guarantees, as RepuCoin relies on a reputation-based Byzantine fault-tolerant consensus. Specifically, the consensus protocol in RepuCoin is deterministic for both keyblocks and microblocks.

Double Spending Attacks. RepuCoin addresses the double spending attack by speeding up the confirmation process to less than a few seconds, even when a block size is as large as 4 MB. In addition, RepuCoin provides a deterministic consistency guarantee rather than a probabilistic one. Determinism is achieved by pinning microblocks and keyblocks through a consensus scheme using a reputation-based voting mechanism. Pinned microblocks and keyblocks are non-reversible, a guarantee provided by our use of consensus.

Eclipse Attacks and Isolated Leaders. In partial (or full) eclipse attacks [4], [5], an attacker capable of delaying information that a victim expects to receive is able to launch double spending attacks and selfish-mining attacks.

RepuCoin does not prevent an attacker from fully isolating a victim or delaying messages from a victim. However, given that blocks are pinned, the attacker cannot successfully launch double spending attacks, as previously explained. In the extreme case, some group members may be isolated temporally due to attacks on the network, e.g., those creating partitions. Such network attacks may delay the block pinning process and prevent RepuCoin from making progress. However, RepuCoin would recover as soon as the messages are delivered, and the attacker can neither create a fork of the blockchain, nor double spend any coin.

7 Conclusion

RepuCoin provides proof-of-reputation as an alternative way to provide a strong deterministic consensus, and be robust against attacks, in a permission-less distributed blockchain system. All BFT-based blockchain systems (e.g., [15], [17], [38], [39], [40]) are bound to the coverage of the assumption on the maximum number of faulty players, f, or their decision power quota thereof. RepuCoin, although belonging to that generation of systems, is the first to deploy effective mitigation measures that reduce brittleness in the face of overwhelming adversary power, where other systems give in. Namely, it provides security guarantees against an attacker who can control a majority of the overall computing power for a duration that increases with the joining time of the attacker.

Based on the strong deterministic guarantee derived from reputation-based weighted voting, the robustness of RepuCoin grows with legitimate operation time: the later the attacker joins, the more secure the system is. For example, an attacker that joins the system after it has been operating for a year, would need at least 51percent of the overall computing power and would need to behave correctly in the system for 10 months before being able to successfully make RepuCoin lose liveness. Breaking RepuCoin's safety is at least as difficult as breaking its liveness. Further discussions on secure bootstrapping, formal security analysis, and applying our PoR to other virtual mining systems (e.g., proof of stake) can be found in our technological report [20].

Acknowledgments

This work is partially supported by the Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009, https://bitcoin.org/bitcoin.pdf
[2] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending fast payments in bitcoin," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 906–917.
[3] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking bitcoin: Routing attacks on cryptocurrencies," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 375–392.
[4] A. Gervais, H. Ritzdorf, G. O. Karame, and S. Capkun, "Tampering with the delivery of blocks and transactions in bitcoin," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 692–705.
[5] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *Proc. 24th USENIX Conf. Secur. Symp.*, 2015, pp. 129–144.
[6] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *Proc. 18th Int. Conf. Financial Cryptography Data Secur.*, 2014, pp. 436–454.
[7] A. Sapirshtein, Y. Sompolinsky, and A. Zohar, "Optimal selfish mining strategies in bitcoin," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2016, pp. 515–532.
[8] J. Bonneau, "Why buy when you can rent? - bribery attacks on bitcoin-style consensus," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2016, pp. 19–26.
[9] Lightning network, 2017. [Online]. Available: https://lightning.network/
[10] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2015. [Online]. Available: tinyurl.com/y9xoa42u
[11] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, et al., "On scaling decentralized blockchains," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2016, pp. 106–125.
[12] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *Proc. 18th Int. Conf. Financial Cryptography Data Secur.*, 2015, pp. 507–527.

PeerMart: The Technology for a Distributed Auction-based Market for Peer-to-Peer Services

David Hausheer[1], Burkhard Stiller[2,1]

[1] Computer Engineering and Networks Laboratory TIK, Swiss Federal Institute of Technology ETH Zurich, Switzerland
[2] Institute for Informatics IFI, University of Zurich, Switzerland

[hausheer|stiller]@tik.ee.ethz.ch
http://www.peermart.net/

***Abstract*-P2P networks are becoming increasingly popular for a wide variety of applications going beyond pure file sharing. However, a commercial use of P2P technology is currently not possible as efficient and reliable market mechanisms are missing. This paper presents PeerMart, a distributed technology in support of a market for trading P2P services. PeerMart combines the economic efficiency of double auctions with the technical efficiency and resilience of structured P2P networks. The system is implemented on top of a redundant P2P infrastructure and is being evaluated with respect to scalability, efficiency, and reliability.**

I. Introduction

Emerging peer-to-peer (P2P) networks implement the idea that peers share resources with other peers without having to rely on a centralized infrastructure. By means of appropriate aggregation and replication techniques P2P systems can provide much higher performance and robustness than traditional client/server-based applications. Today, file sharing systems like eMule [6] or BitTorrent [4] are the most widespread P2P application. However, a number of additional applications have been developed based on P2P technology, like distributed storage, multimedia streaming, or distributed online games, which become increasingly popular.

P2P systems are based on the assumption, that every peer contributes as much as it benefits from other peers. However, as peers are autonomous entities acting in a rational and selfish way [13], it is unlikely that this kind of cooperation will happen in the absence of appropriate economic and social mechanisms. This observation is based on the well-known free-rider problem [1]. Therefore, and without appropriate incentives for peers to cooperate, P2P systems perform very badly as only few peers will offer services or resources.

To alleviate this problem, more and more P2P applications started to adopt accounting mechanisms to enforce balance between contribution and consumption of individual peers, *e.g.*, eMule's credit system [6] or BitTorrent's tit-for-tat mechanism [4]. However, these approaches are mainly file sharing-oriented and do not take into account the value of the services (files) offered. Moreover, it is not possible that credits gained for uploading a file to one peer can be spent for downloading a file from another peer or for using other services.

In order to overcome these shortcomings and enable the commercial use of P2P technology, a complete set of market mechanisms are necessary. PeerMart aims at a generic solution enabling peers to trade any kind of services with potentially different values. An example for such a service could, *e.g.*, be the upload of a file at a particular bandwidth. PeerMart provides the technology for a completely decentralized market for trading such services. In particular, it enables dynamic pricing and efficient price dissemination and lookup for services over a P2P network. Using PeerMart, peers can bid prices for services, which enables them to govern the desired service performance.

The presented approach uses an economically efficient double auction mechanism and combines it with the technically beneficial properties of a structured P2P overlay network such as Chord [15] or Pastry [12]. The core idea is to distribute the broker load of an otherwise centralized auctioneer onto clusters of peers, each being responsible for brokering several services. PeerMart uses the overlay network infrastructure to map the services onto particular sets of peers. By following this fully distributed and redundant approach, a high reliability can be achieved at a relatively low overhead of messages and necessary storage space. In addition, the solution scales very well with the number of participating peers.

The remainder of this paper is organized as follows. Section II describes the basic market model and the main problems which are focussed. In addition, an outline of the technical design space for pricing mechanisms in P2P networks is given. Section III presents the basic design and characteristics of PeerMart, while Section IV further analyzes and evaluates the system in detail. Finally, Section V concludes the paper and gives an outlook on future work. In addition, some further extensions to the basic design will be described.

II. Problem and Design Space

The considered market model for P2P services is depicted in Figure 1. For any service there are basically three different roles, consumers, providers, and intermediate peers acting as brokers. It is assumed that a peer can act in several roles at the same time. While a consumer's goal is to maximize its utility by finding providers offering a particular service at a low price, a provider's goal is to attract consumers that are willing to pay a high price to maximize its benefit for the services offered. Intermediate peers are responsible to match these needs in an efficient and optimal way, i.e. they need to forward

0-7803-8938-7/05/$20.00 (C) 2005 IEEE

messages such as service requests and service offers, as well as process and store service-related information like prices on behalf of other peers.

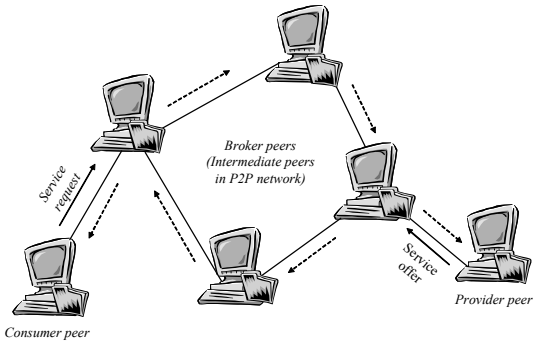


Figure 1. Market Model for P2P Services

A. Problem Statement

One of the main obstacles in a P2P environment is that peers may not be willing to cooperate or behave correctly according to a certain protocol. First of all, cooperation is costly. Therefore, peers may simply have no incentives to cooperate. But even worse, peers can be competitors for a particular service. For example, a provider could lose the opportunity to sell a service. In any case, it has to be taken into account that peers are autonomous entities which act in a rational and selfish way [13].

In general, two different types of selfish behavior can be distinguished, *a)* peers not providing services like files or other resources, and *b)* peers not providing base functionality like forwarding or caching service requests. For the first problem several solutions have been proposed in the past, such as using accounting mechanisms, micro-payment, or reputation-based schemes [7], [9], [16], [18]. However, the second problem appears to be more complicated to deal with, as this type of selfish behavior is generally much harder to detect. For example, it might be quite difficult to track down a peer which did not forward a particular message. The problem becomes even more complex when business sensitive information like price offers need to be processed or stored. Using accounting mechanisms as an incentive to provide such functionality seems not feasible due to the enormous technical effort that would be needed. Also, it might not solve the problem, since the incentive for providing that functionality would need to be higher than the potential benefit for not providing it.

PeerMart attempts to solve both problems in an integrated manner. While the answer to the first problem is to provide an efficient pricing mechanism for P2P services, the second problem is solved by using redundancy, as suggested by [14]. As it will be shown in Section IV, redundancy helps to increase the reliability of PeerMart in the presence of malicious peers.

B. Design Space for Pricing in P2P

There are many alternatives to set prices for services and disseminate them to other peers. The simplest approach would be that fixed prices are determined by a central authority, *e.g.*, the system designer. While such an approach has basically no communication overhead, it implies that the central authority has complete information to set optimal prices in advance, which seems not to be viable in practice.

In the absence of a central authority, the problem becomes technically much more complex. Prices need to be communicated to other peers in a reliable way. A simple approach would be that providers and consumers regularly broadcast price offers for services to all other peers in the network. Any offer could then directly be answered with a counteroffer by any peer. A similar approach is adopted in [5]. Obviously, such a solution does not scale and there are no guarantees whether all peers can be reached. A more scalable but rather complex approach is to store offers for later use in a form of routing table at intermediate peers. Peers could then use this information to route requests to the peer currently offering the best price rather than using broadcast. However, it is unclear how long prices should stay valid before being dropped from such a routing table.

In contrast, auction-based approaches seems to be very promising in terms of both economic and technical efficiency. While single-sided auctions have the drawback of being either provider- or consumer-oriented, double auctions enable both providers and consumers to offer prices. These offers are continuously matched by a broker following a certain matching strategy. Today the double auction is widely used in stock markets. An attempt to implement a double auction in a P2P environment is proposed in [11]. Their algorithm is based on agents which are connected in a random P2P network, and it is considered that only one commodity good is being traded. Agents randomly join to build clusters and assign a single agent as the cluster center which keeps a map of all agents in the cluster. Thus, the maximum cluster size has to be limited which implicates that the solution does not scale well. In addition, it is assumed that messages are never lost or delayed, which seems not to be a realistic assumption for a P2P network in practice.

PeerMart has been designed with a strong focus on the technical feasibility of implementing a reliable double auction mechanism on top of a P2P network. PeerMart's approach is to distribute the broker load onto redundant sets of peers. Its main challenge is to correctly synchronize offers and get to common decisions, given that a certain number of peers acts in a faulty or malicious way. This is similar to the Byzantine Generals Problem [8]. To solve this problem, PeerMart uses public key cryptography to clearly identify the sender of an offer and determines potential matches based on majority decisions among the broker peers.

III. PeerMart Design

In the following, the design of PeerMart's auction mechanism and its implementation on top of a structured P2P overlay network are described in detail.

A. Basic Auction Mechanism

The basic auction mechanism works as follows: A provider (consumer) which is interested in trading a particular service, sends a service offer (service request) to the responsible broker, which is realized by a set of peers as described later on. The broker replies with the current bid price (ask price), which is the current highest buy price (lowest sell price) offered by a peer. Based on this information, the provider (consumer) sends a price offer to the broker, using a particular strategy which can be chosen arbitrarily (a potential bidding strategy that is close to human behavior can be found in [3]). The broker continuously runs the following matching strategy:

a) Upon every price offer received from a provider (consumer), there is no match if the offer is higher (lower) than the current bid price (ask price). The price offer is dropped or stored in a table for later use.

b) If there is a match, the price offer is forwarded to the peer that offered the highest buy price (lowest sell price). The resulting price for the service is set to the mean price between the matching price offers.

B. Underlying Infrastructure

PeerMart implements the described auction mechanism on top of a structured P2P overlay network. Currently, Pastry [12] is applied, but in principle any other P2P overlay infrastructure could be used. The motivation for using Pastry is primarily its Java-based implementation, FreePastry, and its notion of leaf-sets, which represent a set of peers. PeerMart uses Pastry for peers joining or leaving the system, and to find other peers in the network. In Pastry every peer is given a unique 128-bit nodeId, which can be calculated from a peer's IP address or public key using a secure hash function. In PeerMart it is assumed that every peer has a public/private key pair, which is also used to sign and verify messages. A lightweight method that could be adopted to acquire a public/private key pair without having to rely on a public key infrastructure is described in [2]. The method is based on crypto puzzles and limits the rate at which new peers can join the system.

C. System Design

It is assumed that each service being traded over PeerMart can be described by a fixed set of parameters and has a unique serviceId with the same length as a nodeId. For a file service this can, *e.g.*, be achieved by calculating the hash value of the corresponding file. The only considered parameter which can currently vary is the price. The serviceId is mapped onto the address space of the overlay network. The set of n peers (called *broker-set*) which are numerically closest to the serviceId are responsible to act as brokers for that service. It is assumed that the serviceIds are uniformly distributed, thus every peer will on average be responsible for an equal number of services.

Each broker keeps a table for every service it is responsible for. The table has a fixed size of m rows and is used to store at most $m/2$ highest buy prices and $m/2$ lowest sell prices. In addition, the following methods are offered by a broker:

getPrice returns the current bid price (ask price) for a service. If no price offers are available, zero (infinite) is returned.

sendOffer accepts a price offer for a service. It returns true, if the offer could successfully be entered into the table. It returns false, if the price is lower (higher) than the $m/2$-highest buy price ($m/2$-lowest sell price) and therefore had to be dropped.

Furthermore, every provider (consumer) offers the following callback methods:

notifyOffer is called by a broker whenever a price offer matched with another one. The corresponding offers will be removed from the table.

notifyDrop is called when an offer had to be dropped by a broker in a later round.

An example for the double auction mechanism in PeerMart is given in Figure 2. Peer offering or requesting a particular service (with serviceId x), contact the responsible broker-set to get the current bid or ask price and then continuously send their own price offers. Apart from the price, every offer contains a sequence number and a validity time and is signed by the peer's private key. For every peer only the newest offer is kept. The validity time cannot be greater than a certain timeout t. When an offer becomes invalid, it is removed from the table.

Only the first request (to identify the broker peers for a particular service) is routed over the overlay network. All subsequent messages (namely price offer) are sent "directly" over the underlying Internet.

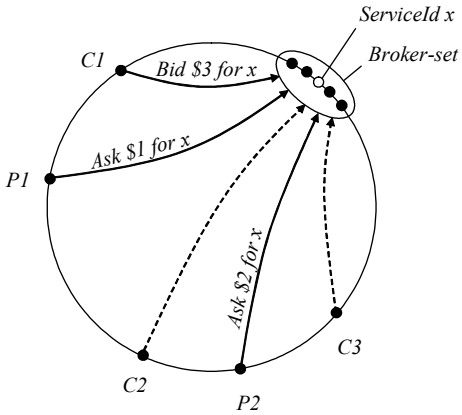


Figure 2. Double Auction in PeerMart

D. Broker-Set Maintenance

When a new service is offered for the first time, the corresponding root node (the node numerically closest to the serviceId) has to notify the other peers in its leaf-set about the new service. If the root node fails to do that (because it is a malicious node), the following fallback method is applied. Recursively, peers on the route to the serviceId are contacted, until the next closest node to the serviceId is found, which has to be in the same leaf-set as the root node. This peer then takes over the notification of the new service. In addition, every broker keeps a list of nodeIds which are in the same broker-set for a

particular service. This list is regularly updated based on changes in the leaf-set notified by Pastry.

E. Broker Redundancy

As a countermeasure against faulty or malicious peers, service offers (requests) and price offers are always sent to f randomly selected broker peers in parallel ($1 \le f \le n$). f is a design parameter that has to be set very carefully with respect to the ratio of malicious peers, desired reliability and message overhead, for which there is always a trade-off. Broker peers that receive an offer either reject it or store it in their tables according to the strategy described above. Every broker peer then forwards pairs of locally matching offers to all other peers in the broker set. Based on the signature of an offer, a broker can easily verify whether the offer is valid. If a broker has no offers that match locally, the current bid price (ask price) is forwarded instead, if it has not already been sent earlier. Based on the forwarded offers from all brokers the current bid price (ask price) is determined and a globally valid matching is performed by every broker. Whenever two offers match, the corresponding peers are notified by the brokers which initially received the offer.

In this redundant approach message loss is implicitly considered. When a message gets lost accidentally, it appears as if the corresponding peers act maliciously.

F. Timing Issues

In PeerMart slotted time is used for every individual service to tackle the problem of message delays. Every offer which potentially matches is delayed for one to two time slots before being forwarded to the other brokers. This approach guarantees that all peers have the same chance to make a deal. Every time slot has a sequence number starting at zero when a service is traded for the first time. Time slots have a fixed duration which is set to the maximum expected round-trip time in the network. During even time slots offers are collected, while during odd time slots potential matches are notified to the other broker peers as described above. Since after this synchronization process all broker peers have the same information needed to match offers, no matching conflict occurs. In the rare case that more than one peers made the same offer within the same time slot, a broker peer forwards the one that came in first. After synchronization, the offer which was forwarded by the most brokers will be selected.

IV. Evaluation

In the following, PeerMart's technical performance is evaluated with respect to efficiency, scalability, and reliability. An analysis of the economic efficiency of the proposed auction mechanism is out of the scope of this paper. Justifications for this fact can be found in [10] and [17].

A. Efficiency and Scalability

The technical efficiency of PeerMart is measured as the amount of overhead in terms of storage space used and messages generated by peers. There is a basic overhead for maintaining the Pastry overlay network. A detailed analysis of the scalability and overhead of Pastry can be found in [12].

Beyond that, PeerMart generates overhead through the exchange, storage, and matching of offers. The number of offers which need to be stored by each peer are limited in two dimensions. First, since the table size is limited, the maximum number of offers per service each broker has to maintain is m. Furthermore, it is assumed that the number of services a peer is concurrently involved in as either a consumer or provider is limited to s, as a peer's resources to consume or provide services are physically bound. Thus, on average every peer has to store a limited number of $s \cdot n \cdot m$ offers, where n is the broker-set size. Note that tables for services in which no peer is involved in anymore (no valid offers) can safely be removed.

The number of messages per service involvement a broker peer needs to deal with is influenced by several factors. Storing/rejecting an offer and notifying a match generates $2 \cdot f$ messages as only a fraction of broker peers are contacted, which will forward the message. When an offer leads to a match, an additional number of $n \cdot f$ messages are generated. A simulation has been performed with a varying number of peers bidding for services. The bidding strategy chosen for the consumers was min(*ask price* + *a**(*bid limit* - *ask price*),*bid limit*) and max(*bid price* - *a**(*bid price* - *ask limit*),*ask limit*) for the providers, respectively, where *a* is the learning parameter which has been set to 0.1. The bid and ask limits (reservation prices) of the peers were normally distributed, such that 50% of the offers were leading to a match. This bidding strategy was motivated by the ZIP strategy proposed in [3].

Figure 3 shows the message overhead per broker peer depending on the number of peers in the network. It can be

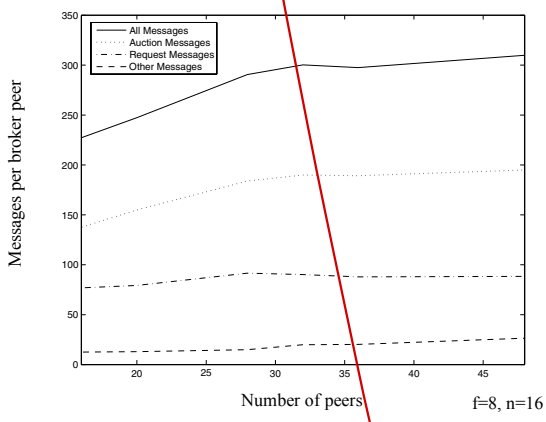


Figure 3. Message Overhead

seen that the total number of messages increases linearly in a small network, but stays almost constant when the network becomes larger. Thus, the system scales very well at a relatively low overhead of messages.

B. Reliability

Reliability denotes the capability to resist against malicious or unreliable peers. As described above, PeerMart uses redun-

dancy to achieve reliability. The redundancy of PeerMart can be configured through the parameter *f*. Thus, the desired reliability can be adjusted based on the expected number of malicious or unreliable peers. Figure 4 shows the reliability of PeerMart (measured in the amount of correctly matched offers), depending on the number of parallel brokers *f* and the amount of malicious peers in the broker set for a fixed broker set size *n*. Malicious peers were modeled as brokers which did not forward offers to other brokers. These are much harder to deal with than peers which are simply offline and can be detected after no reply has been received for a certain time. It can be seen in Figure 4, that with 8 brokers in parallel PeerMart can correctly match offers with up to 50% malicious peers. Thus, a high reliability can be achieved.

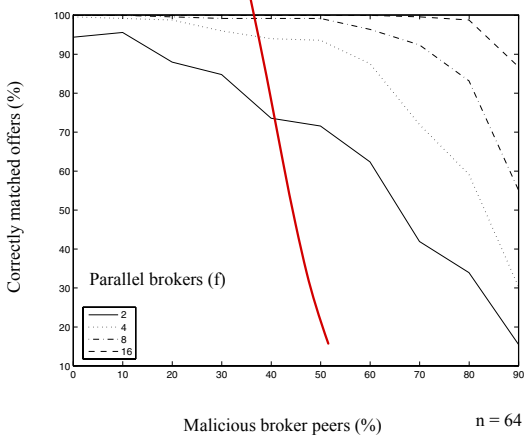


Figure 4. Reliability of PeerMart

V. Conclusions

This paper presented PeerMart, a completely distributed auction-based market based on P2P networks, which can be used by peers to efficiently trade services. Implemented on top of a redundant overlay network infrastructure, PeerMart provides a reliable system, which scales well even for a large number of peers trading services.

In future work PeerMart will be further extended and its efficiency will be optimized. While currently only the price for a service can vary, it will be investigated how several dynamic parameters could be supported.

Furthermore, the following additional types of malicious behavior will be considered:

a) A peer may continuously send offers and ignore any received match. This could be tackled by decreasing the reputation value of a peer whenever it ignores an offer and increasing it upon every accept. If the reputation value falls below a certain threshold, a peer's offers could simply be dropped, so the peer will be excluded from the system.

b) A peer may not stay to its promises, although having previously accepted an offer. This is out of control of PeerMart. Actually, the two involved peers may well further negotiate the price bilaterally, if both peers agree. PeerMart has no means to prevent this. Otherwise, the same reputation mechanism might be used as above.

c) A broker peer may not store or forward an offer. While this is generally not a problem, if at least a few peers behave correctly as shown above, it is still necessary to punish and thus prevent such behavior. A broker peer which does not store an offer cannot immediately be detected. However, if an offer is not forwarded by a broker peer, the receiver will detect it, since fewer than *f* messages are obtained. Again, the reputation value of the malicious peer could be decreased in this case.

Acknowledgement

This work has been performed partially in the framework of the EU IST project MMAPPS "Market Management of Peer-to-Peer Services" (IST-2001-34201), where the ETH Zürich has been funded by the Swiss Bundesministerium für Bildung und Wissenschaft BBW, Bern, under Grant No. 00.0275. Additionally, the authors would like to acknowledge discussions with all of their colleagues and project partners.

References

[1] E. Adar, B. Huberman: *Free Riding on Gnutella;* First Monday, Vol. 5, Nr. 10, October 2000.

[2] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach: *Security for structured peer-to-peer overlay networks*; In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, MA, USA, December 2002.

[3] D. Cliff: *Minimal-Intelligence Agents for Bargaining Behaviors in Market-Based Environments*; Technical Report HPL-97-91, HP Laboratories, Bristol, England, 1997.

[4] B. Cohen: *Incentives Build Robustness in BitTorrent*; Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003.

[5] Z. Despotovic, J. Usunier, K. Aberer: *Towards Peer-To-Peer Double Auctioning*; In Proceedings of the 37th Hawaii International Conference on System Sciences, Waikoloa, HI, USA, January 2004.

[6] The eMule Project: http://www.emule-project.net/.

[7] D. Hausheer, N. Liebau, A. Mauthe, R. Steinmetz, B. Stiller: *Token-based Accounting and Distributed Pricing to Introduce Market Mechanisms in a Peer-to-Peer File Sharing Scenario*; In Proceedings 3rd IEEE International Conference on Peer-to-Peer Computing, Linköping, Sweden, September 2003.

[8] L. Lamport, R. Shostak, M. Pease: *The Byzantine Generals Problem*; ACM Transactions on Programming Languages and Systems, vol. 4, pp. 382-401, July 1982.

[9] T. Moreton, A. Twigg: *Trading in Trust, Tokens, and Stamps*; Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003.

[10] R. Myerson, M. Satterthwaite: *Efficient Mechanisms for Bilateral Trading*; Journal of Economic Theory, Vol. 29, pp. 265-281, 1983.

[11] E. Ogston, S. Vassiliadis: *A Peer-to-Peer Agent Auction*; In Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Bologna, Italy, July 2002.

[12] A. Rowstron and P. Druschel: *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. In Proceedings of IFIP/ACM Middleware 2001, Heidelberg, Germany, November 2001.

[13] J. Shneidman, D. Parkes: *Rationality and Self-Interest in Peer-to-Peer Networks*; 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, USA, February 2003.

[14] J. Shneidman, D. Parkes: *Using Redundancy to Improve Robustness of Distributed Mechanism Implementations*; In Proceedings of 4th ACM Conference on Electronic Commerce (EC'03), San Diego, CA, USA, May 2003.

[15] I. Stoica, R. Morris, D. Karger, M. Kaashoek, H. Balakrishnan: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*; ACM SIGCOMM 2001, pp. 149-160, San Diego, CA, USA, August 2001.

[16] V. Vishnumurthy, S. Chandrakumar, E. G. Sirer: *KARMA : A Secure Economic Framework for Peer-to-Peer Resource*; Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003.

[17] B. Wilson: *Incentive Efficiency of Double Auctions*; Econometrica, Vol. 53, pp. 1101-1115, 1985.

[18] B. Yang, H. Garcia-Molina: *PPay: Micropayments for Peer-to-Peer Systems*; ACM Conference on Computer and Communications Security (CCS '03), Washington, DC, USA, October 2003.

Eclipse Attacks on Bitcoin's Peer-to-Peer Network

Ethan Heilman[*] *Alison Kendler*[*] *Aviv Zohar*[†] *Sharon Goldberg*[*]
[*]*Boston University* [†]*Hebrew University/MSR Israel*

Abstract

We present eclipse attacks on bitcoin's peer-to-peer network. Our attack allows an adversary controlling a sufficient number of IP addresses to monopolize all connections to and from a victim bitcoin node. The attacker can then exploit the victim for attacks on bitcoin's mining and consensus system, including N-confirmation double spending, selfish mining, and adversarial forks in the blockchain. We take a detailed look at bitcoin's peer-to-peer network, and quantify the resources involved in our attack via probabilistic analysis, Monte Carlo simulations, measurements and experiments with live bitcoin nodes. Finally, we present countermeasures, inspired by botnet architectures, that are designed to raise the bar for eclipse attacks while preserving the openness and decentralization of bitcoin's current network architecture.

1 Introduction

While cryptocurrency has been studied since the 1980s [22, 25, 28], bitcoin is the first to see widespread adoption.A key reason for bitcoin's success is its baked-in decentralization. Instead of using a central bank to regulate currency, bitcoin uses a decentralized network of nodes that use computational proofs-of-work to reach consensus on a distributed public ledger of transactions, *aka.,* the *blockchain*. Satoshi Nakamoto [52] argues that bitcoin is secure against attackers that seek to shift the blockchain to an inconsistent/incorrect state, as long as these attackers control less than half of the computational power in the network. But underlying this security analysis is the crucial assumption of *perfect information*; namely, that all members of the bitcoin ecosystem can observe the proofs-of-work done by their peers.

While the last few years have seen extensive research into the security of bitcoin's computational proof-of-work protocol *e.g.,* [14, 29, 36, 37, 45, 49, 50, 52, 58, 60], less attention has been paid to the peer-to-peer network used to broadcast information between bitcoin nodes (see Section 8). The bitcoin peer-to-peer network, which is bundled into the core bitcoind implementation, *aka.,* the Satoshi client, is designed to be open, decentralized, and independent of a public-key infrastructure. As such, cryptographic authentication between peers is not used, and nodes are identified by their IP addresses (Section 2). Each node uses a randomized protocol to select eight peers with which it forms long-lived *outgoing connections*, and to propagate and store addresses of other potential peers in the network. Nodes with public IPs also accept up to 117 *unsolicited incoming connections* from any IP address. Nodes exchange views of the state of the blockchain with their incoming and outgoing peers.

Eclipse attacks. This openness, however, also makes it possible for adversarial nodes to join and attack the peer-to-peer network. In this paper, we present and quantify the resources required for *eclipse attacks* on nodes with public IPs running bitcoind version 0.9.3. In an eclipse attack [27, 61, 62], the attacker monopolizes all of the victim's incoming and outgoing connections, thus isolating the victim from the rest of its peers in the network. The attacker can then filter the victim's view of the blockchain, force the victim to waste compute power on obsolete views of the blockchain, or coopt the victim's compute power for its own nefarious purposes (Section 1.1). We present *off-path* attacks, where the attacker controls endhosts, but not key network infrastructure between the victim and the rest of the bitcoin network. Our attack involves rapidly and repeatedly forming unsolicited incoming connections to the victim from a set of endhosts at attacker-controlled IP addresses, sending bogus network information, and waiting until the victim restarts (Section 3). With high probability, the victim then forms all eight of its outgoing connections to attacker-controlled addresses, and the attacker also monopolizes the victim's 117 incoming connections.

Our eclipse attack uses extremely low-rate TCP connections, so the main challenge for the attacker is to

obtain a sufficient number of IP addresses (Section 4). We consider two attack types: (1) infrastructure attacks, modeling the threat of an ISP, company, or nation-state that holds several *contiguous* IP address blocks and seeks to subvert bitcoin by attacking its peer-to-peer network, and (2) botnet attacks, launched by bots with addresses in *diverse* IP address ranges. We use probabilistic analysis, (Section 4) measurements (Section 5), and experiments on our own live bitcoin nodes (Section 6) to find that while botnet attacks require far fewer IP addresses, there are hundreds of organizations that have sufficient IP resources to launch eclipse attacks (Section 4.2.1). For example, we show how an infrastructure attacker with 32 distinct /24 IP address blocks (8192 address total), or a botnet of 4600 bots, can always eclipse a victim with at least 85% probability; this is independent of the number of nodes in the network. Moreover, 400 bots sufficed in tests on our live bitcoin nodes. To put this in context, if 8192 attack nodes joined today's network (containing $\approx$ 7200 public-IP nodes [4]) and honestly followed the peer-to-peer protocol, they could eclipse a target with probability about $(\frac{8192}{7200+8192})^8 = 0.6\%$.

Our attack is only for nodes with public IPs; nodes with private IPs may be affected if all of their outgoing connections are to eclipsed public-IP nodes.

Countermeasures. Large miners, merchant clients and online wallets have been known to modify bitcoin's networking code to reduce the risk of network-based attacks. Two countermeasures are typically recommended [3]: (1) disabling incoming connections, and (2) choosing 'specific' outgoing connections to well-connected peers or known miners (*i.e.,* use whitelists). However, there are several problems with scaling this to the full bitcoin network. First, if incoming connections are banned, how do new nodes join the network? Second, how does one decide which 'specific' peers to connect to? Should bitcoin nodes form a private network? If so, how do they ensure compute power is sufficiently decentralized to prevent mining attacks?

Indeed, if bitcoin is to live up to its promise as an open and decentralized cryptocurrency, we believe its peer-to-peer network should be open and decentralized as well. Thus, our next contribution is a set of countermeasures that preserve openness by allowing unsolicited incoming connections, while raising the bar for eclipse attacks (Section 7). Today, an attacker with enough addresses can eclipse *any* victim that accepts incoming connections and then restarts. Our countermeasures ensure that, with high probability, if a victim stores enough legitimate addresses that accept incoming connections, then the victim be cannot eclipsed *regardless of the number of IP addresses the attacker controls.* Our countermeasures 1, 2, and 6 have been deployed in bitcoind v0.10.1; we also developed a patch [40] with Countermeasures 3,4.

1.1 Implications of eclipse attacks

Apart from disrupting the bitcoin network or selectively filtering a victim's view of the blockchain, eclipse attacks are a useful building block for other attacks.

Engineering block races. A block race occurs when two miners discover blocks at the same time; one block will become part of the blockchain, while the other "orphan block" will be ignored, yielding no mining rewards for the miner that discovered it. An attacker that eclipses many miners can engineer block races by hording blocks discovered by eclipsed miners, and releasing blocks to both the eclipsed and non-eclipsed miners once a competing block has been found. Thus, the eclipsed miners waste effort on orphan blocks.

Splitting mining power. Eclipsing an x-fraction of miners eliminates their mining power from the rest of the network, making it easier to launch mining attacks (*e.g.,* the 51% attack [52]). To hide the change in mining power under natural variations [19], miners could be eclipsed gradually or intermittently.

Selfish mining. With selfish mining [14,29,37,60], the attacker strategically withholds blocks to win more than its fair share of mining rewards. The attack's success is parameterized by two values: α, the ratio of mining power controlled by the attacker, and γ, the ratio of honest mining power that will mine on the attacker's blocks during a block race. If γ is large, then α can be small. By eclipsing miners, the attacker increases γ, and thus decreases α so that selfish mining is easier. To do this, the attacker drops any blocks discovered by eclipsed miners that compete with the blocks discovered by the selfish miners. Next, the attacker increases γ by feeding only the selfish miner's view of the blockchain to the eclipsed miner; this coopts the eclipsed miner's compute power, using it to mine on the selfish-miner's blockchain.

Attacks on miners can harm the entire bitcoin ecosystem; mining pools are also vulnerable if their gateways to the public bitcoin network can be eclipsed. Eclipsing can also be used for double-spend attacks on non-miners, where the attacker spends some bitcoins multiple times:

0-confirmation double spend. In a 0-confirmation transaction, a customer pays a transaction to a merchant who releases goods to the customer *before* seeing a block confirmation *i.e.,* seeing the transaction in the blockchain [18]. These transactions are used when it is inappropriate to wait the 5-10 minutes typically needed to for a block confirmation [20], *e.g.,* in retail point-of-sale systems like BitPay [5], or online gambling sites like Betcoin [57]. To launch a double-spend attack against the merchant [46], the attacker eclipses the merchant's bitcoin node, sends the merchant a transaction T for goods, and sends transaction T' double-spending those

bitcoins to the rest of the network. The merchant releases the goods to the attacker, but since the attacker controls all of the merchant's connections, the merchant cannot tell the rest of the network about T, which meanwhile confirms T'. The attacker thus obtains the goods without paying. 0-confirmation double-spends have occurred in the wild [57]. This attack is as effective as a Finney attack [39], but uses eclipsing instead of mining power.

***N*-confirmation double spend.** If the attacker has eclipsed an x-fraction of miners, it can also launch N-confirmation double-spending attacks on an eclipsed merchant. In an N-confirmation transaction, a merchant releases goods only after the transaction is confirmed in a block of depth $N-1$ in the blockchain [18]. The attacker sends its transaction to the eclipsed miners, who incorporate it into their (obsolete) view of the blockchain. The attacker then shows this view of blockchain to the eclipsed merchant, receives the goods, and sends both the merchant and eclipsed miners the (non-obsolete) view of blockchain from the non-eclipsed miners. The eclipsed miners' blockchain is orphaned, and the attacker obtains goods without paying. This is similar to an attack launched by a mining pool [10], but our attacker eclipses miners instead of using his own mining power.

Other attacks exist, *e.g.,* a transaction hiding attack on nodes running in SPV mode [16].

2 Bitcoin's Peer-to-Peer Network

We now describe bitcoin's peer-to-peer network, based on bitcoind version 0.9.3, the most current release from 9/27/2014 to 2/16/2015, whose networking code was largely unchanged since 2013. This client was originally written by Satoshi Nakamoto, and has near universal market share for public-IP nodes (97% of public-IP nodes according to Bitnode.io on 2/11/2015 [4]).

Peers in the bitcoin network are identified by their IP addresses. A node with a public IP can initiate up to *eight outgoing connections* with other bitcoin nodes, and accept up to 117 *incoming connections*.[1] A node with a private IP only initiates eight outgoing connections. Connections are over TCP. Nodes only propagate and store public IPs; a node can determine if its peer has a public IP by comparing the IP packet header with the bitcoin `VERSION` message. A node can also connect via Tor; we do not study this, see [16, 17] instead. We now describe how nodes propagate and store network information, and how they select outgoing connections.

[1] This is a configurable. Our analysis only assumes that nodes have 8 outgoing connections, which was confirmed by [51]'s measurements.

2.1 Propagating network information

Network information propagates through the bitcoin network via DNS seeders and `ADDR` messages.

DNS seeders. A DNS seeder is a server that responds to DNS queries from bitcoin nodes with a (not cryptographically-authenticated) list of IP addresses for bitcoin nodes. The seeder obtains these addresses by periodically crawling the bitcoin network. The bitcoin network has six seeders which are queried in two cases only. The first when a new node joins the network for the first time; it tries to connect to the seeders to get a list of active IPs, and otherwise fails over to a hardcoded list of about 600 IP addresses. The second is when an existing node restarts and reconnects to new peers; here, the seeder is queried only if 11 seconds have elapsed since the node began attempting to establish connections and the node has less than two outgoing connections.

`ADDR` messages. `ADDR` messages, containing up to 1000 IP address and their timestamps, are used to obtain network information from peers. Nodes accept unsolicited `ADDR` messages. An `ADDR` message is solicited *only* upon establishing a outgoing connection with a peer; the peer responds with up to three `ADDR` message each containing up to 1000 addresses randomly selected from its tables. Nodes push `ADDR` messages to peers in two cases. Each day, a node sends its own IP address in a `ADDR` message to each peer. Also, when a node receives an `ADDR` message with no more than 10 addresses, it forwards the `ADDR` message to two randomly-selected connected peers.

2.2 Storing network information

Public IPs are stored in a node's `tried` and `new` tables. Tables are stored on disk and persist when a node restarts.

The `tried` table. The `tried` table consists of 64 *buckets*, each of which can store up to 64 unique addresses for peers to whom the node has successfully established an incoming or outgoing connection. Along with each stored peer's address, the node keeps the timestamp for the most recent successful connection to this peer.

Each peer's address is mapped to a bucket in `tried` by taking the hash of the peer's (a) IP address and (b) *group*, where the group defined is the /16 IPv4 prefix containing the peer's IP address.A bucket is selected as follows:

```
SK = random value chosen when node is born.
IP = the peer's IP address and port number.
Group = the peer's group

i = Hash( SK, IP ) % 4
Bucket = Hash( SK, Group, i ) % 64
return Bucket
```

Thus, every IP address maps to a single bucket in `tried`, and each group maps to up to four buckets.

When a node successfully connects to a peer, the peer's address is inserted into the appropriate `tried` bucket. If the bucket is full (*i.e.,* contains 64 addresses), then *bitcoin eviction* is used: four addresses are randomly selected from the bucket, and the oldest is (1) replaced by the new peer's address in `tried`, and then (2) inserted into the `new` table. If the peer's address is already present in the bucket, the timestamp associated with the peer's address is updated. The timestamp is also updated when an actively connected peer sends a `VERSION`, `ADDR`, `INVENTORY`, `GETDATA` or `PING` message and more than 20 minutes elapsed since the last update.

The `new` table. The `new` table consists of 256 buckets, each of which can hold up 64 addresses for peers to whom the node has not yet initiated a successful connection. A node populates the `new` table with information learned from the DNS seeders, or from `ADDR` messages.

Every address a inserted in `new` belongs to (1) a *group*, defined in our description of the `tried` table, and (2) a *source group*, the group the contains the IP address of the connected peer or DNS seeder from which the node learned address a. The bucket is selected as follows:

```
SK = random value chosen when node is born.
Group     = /16 containing IP to be inserted.
Src_Group = /16 containing IP of peer sending IP.

i = Hash( SK, Src_Group, Group ) % 32
Bucket = Hash( SK, Src_Group, i) % 256
return Bucket
```

Each (*group*, *source group*) pair hashes to a single `new` bucket, while each *group* selects up to 32 buckets in `new`. Each bucket holds unique addresses. If a bucket is full, then a function called `isTerrible` is run over all 64 addresses in the bucket; if any one of the addresses is terrible, in that it is (a) more than 30 days old, or (b) has had too many failed connection attempts, then the terrible address is evicted in favor of the new address; otherwise, *bitcoin eviction* is used with the small change that the evicted address is discarded.

2.3 Selecting peers

New outgoing connections are selected if a node restarts or if an outgoing connection is dropped by the network. A bitcoin node never deliberately drops a connection, except when a blacklisting condition is met (*e.g.,* the peer sends `ADDR` messages that are too large).

A node with $\omega \in [0,7]$ outgoing connections selects the $\omega+1^{th}$ connection as follows:

(1) Decide whether to select from `tried` or `new`, where

$$\Pr[\text{Select from } \texttt{tried}] = \frac{\sqrt{\rho}(9-\omega)}{(\omega+1)+\sqrt{\rho}(9-\omega)} \quad (1)$$

and ρ is the ratio between the number of addresses stored in `tried` and the number of addresses stored in `new`.

(2) Select a random address from the table, with a bias towards addresses with fresher timestamps: (i) Choose a random non-empty bucket in the table. (ii) Choose a random position in that bucket. (ii) If there is an address at that position, return the address with probability

$$p(r,\tau) = \min(1, \tfrac{1.2^r}{1+\tau}) \quad (2)$$

else, reject the address and return to (i). The acceptance probability $p(r,\tau)$ is a function of r, the number of addresses that have been rejected so far, and τ, the difference between the address's timestamp and the current time in measured in ten minute increments.[2]

(3) Connect to the address. If connection fails, go to (1).

3 The Eclipse Attack

Our attack is for a victim with a public IP. Our attacker (1) populates the `tried` table with addresses for its attack nodes, and (2) overwrites addresses in the `new` table with "trash" IP addresses that are not part of the bitcoin network. The "trash" addresses are unallocated (*e.g.,* listed as "available" by [56]) or as "reserved for future use" by [43] (*e.g.,* 252.0.0.0/8). We fill `new` with "trash" because, unlike attacker addresses, "trash" is not a scarce resource. The attack continues until (3) the victim node restarts and chooses new outgoing connections from the `tried` and `new` tables in its persistant storage (Section 2.3). With high probability, the victim establishes all eight outgoing connections to attacker addresses; all eight addresses will be from `tried`, since the victim cannot connect to the "trash" in `new`. Finally, the attacker (5) occupies the victim's remaining 117 incoming connections. We now detail each step of our attack.

3.1 Populating `tried` and `new`

The attacker exploits the following to fill `tried` and `new`:

1. Addresses from unsolicited incoming connections are stored in the `tried` table; thus, the attacker can insert an address into the victim's `tried` table simply by connecting to the victim from that address. Moreover, the *bitcoin eviction* discipline means that the attacker's fresher addresses are likely to evict any older legitimate addresses stored in the `tried` table (Section 2.2).

2. A node accepts unsolicited `ADDR` messages; these addresses are inserted directly into the `new` table without testing their connectivity (Section 2.2). Thus, when our attacker connects to the victim from an adversarial address, it can also send `ADDR` messages with 1000 "trash"

[2] The algorithm also considers the number of failed connections to this address; we omit this because it does not affect our analysis.

addresses. Eventually, the trash overwrites all legitimate addresses in `new`. We use "trash" because we do not want to waste our IP address resources on overwriting `new`.

3. Nodes only rarely solicit network information from peers and DNS seeders (Section 2.1). Thus, while the attacker overwrites the victim's `tried` and `new` tables, the victim almost never counteracts the flood of adversarial information by querying legitimate peers or seeders.

3.2 Restarting the victim

Our attack requires the victim to restart so it can connect to adversarial addresses. There are several reasons why a bitcoin node could restart, including ISP outages, power failures, and upgrades, failures or attacks on the host OS; indeed, [16] found that a node with a public IP has a 25% chance of going offline after 10 hours. Another predictable reason to restart is a software update; on 1/10/2014, for example, bitnodes.io saw 942 nodes running Satoshi client version 0.9.3, and by 29/12/2014, that number had risen to 3018 nodes, corresponding to over 2000 restarts. Since updating is often *not* optional, especially when it corresponds to critical security issues; 2013 saw three such bitcoin upgrades, and the heartbleed bug [53] caused one in 2014. Also, since the community needs to be notified about an upgrade in advance, the attacker could watch for notifications and then commence its attack [2]. Restarts can also be deliberately elicited via DDoS [47, 65], memory exhaustion [16], or packets-of-death (which have been found for bitcoind [6, 7]). The bottom line is that the security of the peer-to-peer network should not rely on 100% node uptime.

3.3 Selecting outgoing connections

Our attack succeeds if, upon restart, the victim makes all its outgoing connections to attacker addresses. To do this, we exploit the bias towards selecting addresses with fresh timestamps from `tried`; by investing extra time into the attack, our attacker ensures its addresses are fresh, while all legitimate addresses become increasingly stale. We analyze this with few simple assumptions:

1. An f-fraction of the addresses in the victim's `tried` table are controlled by the adversary and the remaining $1-f$-fraction are legitimate. (Section 4 analyzes how many addresses the adversary therefore must control.)

2. *All* addresses in `new` are "trash"; all connections to addresses in `new` fail, and the victim is forced to connect to addresses from `tried` (Section 2.3).

3. The attack proceeds in *rounds*, and repeats each round until the moment that the victim restarts. During a single round, the attacker connects to the victim from each of its adversarial IP addresses. A round takes time τ_a, so all adversarial addresses in `tried` are younger than τ_a.

4. An f'-fraction addresses in `tried` are actively connected to the victim before the victim restarts. The timestamps on these legitimate addresses are updated every 20 minute or more (Section 2.2). We assume these timestamps are fresh (*i.e.,* $\tau = 0$) when the victim restarts; this is the worst case for the attacker.

5. The *time invested in the attack* τ_ℓ is the time elapsed from the moment the adversary starts the attack, until the victim restarts. If the victim did not obtain new legitimate network information during of the attack, then, excluding the f'-fraction described above, the legitimate addresses in `tried` are older than τ_ℓ.

Success probability. If the adversary owns an f-fraction of the addresses in `tried`, the probability that an adversarial address is accepted on the first try is $p(1,\tau_a)\cdot f$ where $p(1,\tau_a)$ is as in equation (2); here we use the fact that the adversary's addresses are no older than τ_a, the length of the round. If $r-1$ addresses were rejected during this attempt to select an address from `tried`, then the probability that an adversarial address is accepted on the r^{th} try is bounded by

$$p(r,\tau_a)\cdot f\prod_{i=1}^{r-1} g(i,f,f',\tau_a,\tau_\ell)$$

where

$$g(i,f,f',\tau_a,\tau_\ell) = (1-p(i,\tau_a))\cdot f + (1-p(i,0))\cdot f' + (1-p(i,\tau_\ell))\cdot(1-f-f')$$

is the probability that an address was rejected on the i^{th} try given that it was also rejected on the $i-1^{th}$ try. An adversarial address is thus accepted with probability

$$q(f,f',\tau_a,\tau_\ell) = \sum_{r=1}^{\infty} p(r,\tau_a)\cdot f\prod_{i=1}^{r-1} g(i,f,f',\tau_a,\tau_\ell) \tag{3}$$

and the victim is eclipsed if all eight outgoing connections are to adversarial addresses, which happens with probability $q(f,f',\tau_a,\tau_\ell)^8$. Figure 1 plots $q(f,f',\tau_a,\tau_\ell)^8$ vs f for $\tau_a = 27$ minutes and different choices of τ_ℓ; we assume that $f' = \frac{8}{64\times 64}$, which corresponds to a full `tried` table containing eight addresses that are actively connected before the victim restarts.

Random selection. Figure 1 also shows success probability if addresses were just selected uniformly at random from each table. We do this by plotting f^8 vs f. Without random selection, the adversary has a 90% success probability even if it only fills $f = 72\%$ of `tried`, as long as it attacks for $\tau_\ell = 48$ hours with $\tau_a = 27$ minute rounds. With random selection, 90% success probability requires $f = 98.7\%$ of `tried` to be attacker addresses.

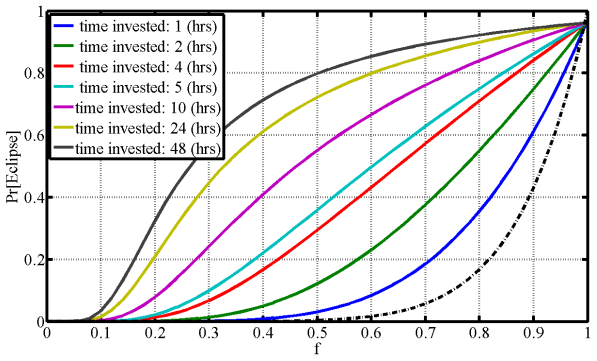


Figure 1: Probability of eclipsing a node $q(f, f', \tau_a, \tau_\ell)$[8] (equation (3)) vs f the fraction of adversarial addresses in `tried`, for different values of time invested in the attack τ_ℓ. Round length is $\tau_a = 27$ minutes, and $f' = \frac{8}{64\times 64}$. The dotted line shows the probability of eclipsing a node if random selection is used instead.

3.4 Monopolizing the eclipsed victim

Figure 1 assumes that the victim has exactly eight *outgoing connections*; all we require in terms of *incoming connections* is that the victim has a few open slots to accept incoming TCP connections from the attacker.

While it is often assumed that the number of TCP connections a computer can make is limited by the OS or the number of source ports, this applies only when OS-provided TCP sockets are used; a dedicated attacker can open an arbitrary number of TCP connections using a custom TCP stack. A custom TCP stack (see *e.g.,* zmap [35]) requires minimal CPU and memory, and is typically bottlenecked only by bandwidth, and the bandwidth cost of our attack is minimal:

Attack connections. To fill the `tried` table, our attacker repeatedly connects to the victim from each of its addresses. Each connection consists of a TCP handshake, bitcoin `VERSION` message, and then disconnection via TCP RST; this costs 371 bytes upstream and 377 bytes downstream. Some attack connections also send one `ADDR` message containing 1000 addresses; these `ADDR` messages cost 120087 bytes upstream and 437 bytes downstream including TCP ACKs.

Monopolizing connections. If that attack succeeds, the victim has eight outgoing connections to the attack nodes, and the attacker must occupy the victim's remaining incoming connections. To prevent others from connecting to the victim, these TCP connections could be maintained for 30 days, at which point the victim's address is `terrible` and forgotten by the network. While bitcoin supports block inventory requests and the sending of blocks and transactions, this consumes significant bandwidth; our attacker thus does not to respond to inventory requests. As such, setting up each TCP connection costs 377 bytes upstream and 377 bytes downstream, and is maintained by ping-pong packets and TCP ACKs consuming 164 bytes every 80 minutes.

We experimentally confirmed that a bitcoin node will accept all incoming connections from the same IP address. (We presume this is done to allow multiple nodes behind a NAT to connect to the same node.) Maintaining the default 117 incoming TCP connections costs $\frac{164\times 117}{80\times 60} \approx 4$ bytes per second, easily allowing one computer to monopolize multiple victims at the same time. As an aside, this also allows for *connection starvation attacks* [32], where an attacker monopolizes all the incoming connections in the peer-to-peer network, making it impossible for new nodes to connect to new peers.

4 How Many Attack Addresses?

Section 3.3 showed that the success of our attack depends heavily on τ_ℓ, the time invested in the attack, and f, the fraction of attacker addresses in the victim's `tried` table. We now use probabilistic analysis to determine how many addresses the attacker must control for a given value of f; it's important to remember, however, that even if f is small, our attacker can still succeed by increasing τ_ℓ. Recall from Section 2.2 that bitcoin is careful to ensure that a node does not store too many IP addresses from the same *group* (*i.e.,* /16 IPv4 address block). We therefore consider two attack variants:

Botnet attack (Section 4.1). The attacker holds several IP addresses, each in a *distinct* group. This models attacks by a botnet of hosts scattered in diverse IP address blocks. Section 4.1.1 explains why many botnets have enough IP address diversity for this attack.

Infrastructure attack (Section 4.2). The attacker controls several IP address blocks, and can intercept bitcoin traffic sent to any IP address in the block, *i.e.,* the attacker holds multiple sets of addresses in the same *group*. This models a company or nation-state that seeks to undermine bitcoin by attacking its network. Section 4.2.1 discusses organizations that can launch this attack.

We focus here on `tried`; Appendix B considers how to send "trash"-filled `ADDR` messages that overwrite `new`.

4.1 Botnet attack

The botnet attacker holds t addresses in distinct groups. We model each address as hashing to a uniformly-random bucket in `tried`, so the number of addresses hashing to each bucket is binomally distributed[3] as $B(t, \frac{1}{64})$. How many of the 64×64 entries in `tried`

[3] $B(n, p)$ is a binomial distribution counting successes in a sequence of n independent yes/no trials, each yielding 'yes' with probability p.

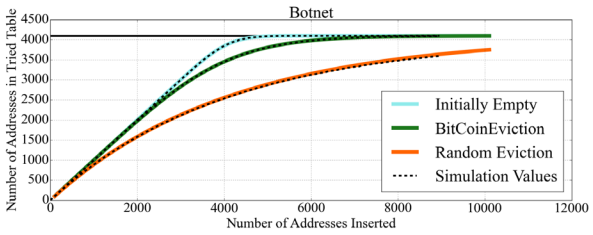


Figure 2: Botnet attack: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses (bots) t. Values were computed from equations (4), (7) and (8), and confirmed by Monte Carlo simulations (with 100 trials/data point).

can the attacker occupy? We model various scenarios, and plot results in Figure 2.

1. Initially empty. In the best case for the attacker, all 64 buckets are initially empty and the expected number of adversarial addresses stored in the `tried` table is

$$64E[\min(64, B(t, \tfrac{1}{64}))] \tag{4}$$

2. Bitcoin eviction. Now consider the worst case for the attacker, where each bucket i is full of 64 legitimate addresses. These addresses, however, will be *older* than all A_i distinct adversarial addresses that the adversary attempts to insert into to bucket i. Since the bitcoin eviction discipline requires each newly inserted address to select four random addresses stored in the bucket and to evict the oldest, if one of the four selected addresses is a legitimate address (which will be older than all of the adversary's addresses), the legitimate address will be overwritten by the adversarial addresses.

For $a = 0....A_i$, let Y_a be the number of adversarial addresses actually stored in bucket i, given that the adversary inserted a unique addresses into bucket i. Let $X_a = 1$ if the a^{th} inserted address successfully overwrites a legitimate address, and $X_a = 0$ otherwise. Then,

$$E[X_a|Y_{a-1}] = 1 - (\tfrac{Y_{a-1}}{64})^4$$

and it follows that

$$E[Y_a|Y_{a-1}] = Y_{a-1} + 1 - (\tfrac{Y_{a-1}}{64})^4 \tag{5}$$

$$E[Y_1] = 1 \tag{6}$$

where (6) follows because the bucket is initially full of legitimate addresses. We now have a recurrence relation for $E[Y_a]$, which we can solve numerically. The expected number of adversarial addresses in all buckets is thus

$$64 \sum_{a=1}^{t} E[Y_a] \Pr[B(t, \tfrac{1}{64}) = a] \tag{7}$$

3. Random eviction. We again consider the attacker's worst case, where each bucket is full of legitimate addresses, but now we assume that each inserted address evicts a randomly-selected address. (This is not what bitcoin does, but we analyze it for comparison.) Applying Lemma A.1 (Appendix A) we find the expected number of adversarial addresses in all buckets is

$$4096(1 - (\tfrac{4095}{4096})^t) \tag{8}$$

4. Exploiting multiple rounds. Our eclipse attack proceeds in *rounds*; in each round the attacker repeatedly inserts each of his t addresses into the `tried` table. While each address always maps to the same bucket in `tried` in each round, bitcoin eviction maps each address to a *different slot* in that bucket in every round. Thus, an adversarial address that is not stored into its `tried` bucket at the end of one round, might still be successfully stored into that bucket in a future round. Thus far, this section has only considered a single round. But, more addresses can be stored in `tried` by repeating the attack for multiple rounds. After sufficient rounds, the expected number of addresses is given by equation (4), *i.e.,* the attack performs as in the best-case for the attacker!

4.1.1 Who can launch a botnet attack?

The 'initially empty' line in Figure 2 indicates that a botnet exploiting multiple rounds can completely fill `tried` with ≈ 6000 addresses. While such an attack cannot easily be launched from a legitimate cloud service (which typically allocates < 20 addresses per tenant [1, 8, 9]), botnets of this size and larger than this have attacked bitcoin [45, 47, 65]; the Miner botnet, for example, had 29,000 hosts with public IPs [54]. While some botnet infestations concentrate in a few IP address ranges [63], it is important to remember that our botnet attack requires no more than ≈ 6000 groups; many botnets are orders of magnitude larger [59]. For example, the Walowdac botnet was mostly in ranges 58.x-100.x and 188.x-233.x [63], which creates $42 \times 2^8 + 55 \times 2^8 = 24832$ groups. Randomly sampling from the list of hosts in the Carna botnet [26] 5000 times, we find that 1250 bots gives on average 402 distinct groups, enough to attack our live bitcoin nodes (Section 6). Furthermore, we soon show in Figure 3 that an infrastructure attack with $s > 200$ groups easily fills every bucket in `tried`; thus, with $s > 400$ groups, the attack performs as in Figure 2, even if many bots are in the same group. .

4.2 Infrastructure attack

The attacker holds addresses in s distinct *groups*. We determine how much of `tried` can be filled by an attacker controlling s groups s containing t IP addresses/group.

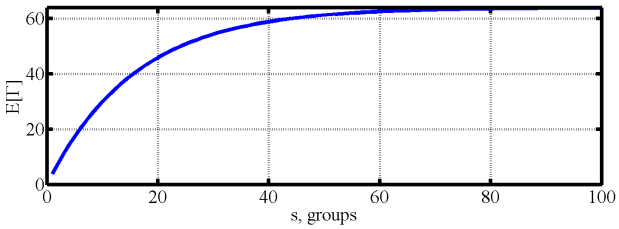


Figure 3: Infrastructure attack. $E[\Gamma]$ (expected number of non-empty buckets) in `tried` vs s (number of groups).

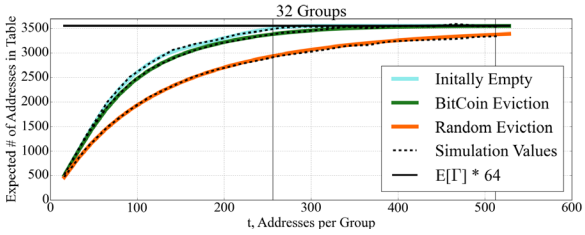


Figure 4: Infrastructure attack with $s = 32$ groups: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses per group t. Results obtained by taking the product of equation (9) and equations from the full version [41], and confirmed by Monte Carlo simulations (100 trials/data point). The horizontal line assumes all $E[\Gamma]$ buckets per (9) are full.

How many groups? We model the process of populating `tried` (per Section 2.2) by supposing that four independent hash functions map each of the s groups to one of 64 buckets in `tried`. Thus, if $\Gamma \in [0,64]$ counts the number of non-empty buckets in `tried`, we use Lemma A.1 to find that

$$E[\Gamma] = 64\left(1-\left(\tfrac{63}{64}\right)^{4s}\right) \approx \left(1-e^{-\frac{4s}{64}}\right) \qquad (9)$$

Figure 3 plots $E[\Gamma]$; we expect to fill 55.5 of 64 buckets with $s = 32$, and all but one bucket with $s > 67$ groups.

How full is the `tried` table? The full version [41] determines the expected number of addresses stored per bucket for the first three scenarios described in Section 4.1; the expected fraction $E[f]$ of `tried` filled by adversarial addresses is plotted in in Figure 4. The horizontal line in Figure 4 show what happens if each of $E[\Gamma]$ buckets per equation (9) is full of attack addresses.

The adversary's task is easiest when all buckets are initially empty, or when a sufficient number of rounds are used; a single /24 address block of 256 addresses suffices to fill each bucket when $s = 32$ grouips is used. Moreover, as in Section 4.1, an attack that exploits multiple rounds performs as in the 'initially empty' scenario. Concretely, with 32 groups of 256 addresses each (8192 addresses in total) an adversary can expect to fill about $f = 86\%$ of the `tried` table after a sufficient number of

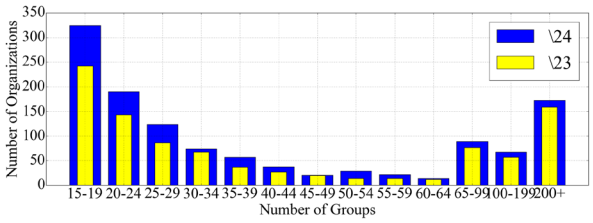


Figure 5: Histogram of the number of organizations with s groups. For the /24 data, we require $t = 256$ addresses per group; for /23, we require $t = 512$.

rounds. The attacker is almost as effective in the bitcoin-eviction scenario with only one round; meanwhile, one round is much less effective with random eviction.

4.2.1 Who can launch an infrastructure attack?

Which organizations have enough IP address resources to launch infrastructure attacks? We compiled data mapping IPv4 address allocation to organizations, using CAIDA's AS to organization dataset [23] and AS to prefix dataset [24] from July 2014, supplementing our data with information from the RIPE database [55]. We determined how many groups (*i.e.,* addresses in the same /16 IPv4 address block) and addresses per group are allocated to each organization; see Figure 5. There are 448 organizations with over $s = 32$ groups and at least $t = 256$ addresses per group; if these organizations invest $\tau_\ell = 5$ hours into an attack with a $\tau_a = 27$-minute round, then they eclipse the victim with probability greater than 80%.

National ISPs in various countries hold a sufficient number of groups ($s \geq 32$) for this purpose; for example, in Sudan (Sudanese Mobile), Columbia (ETB), UAE (Etisalat), Guatemala (Telgua), Tunisia (Tunisia Telecom), Saudi Arabia (Saudi Telecom Company) and Dominica (Cable and Wireless). The United States Department of the Interior has enough groups ($s = 35$), as does the S. Korean Ministry of Information and Communication ($s = 41$), as do hundreds of others.

4.3 Summary: infrastructure or botnet?

Figures 4, 2 show that the botnet attack is far superior to the infrastructure attack. Filling $f = 98\%$ of the victim's `tried` table requires a 4600 node botnet (attacking for a sufficient number of rounds, per equation (4)). By contrast, an infrastructure attacker needs 16,000 addresses, consisting of $s = 63$ groups (equation (9)) with $t = 256$ addresses per group. However, per Section 3.3, if our attacker increases the time invested in the attack τ_ℓ, it can be far less aggressive about filling `tried`. For example, per Figure 1, attacking for $\tau_\ell = 24$ hours with $\tau_a = 27$ minute rounds, our success probability exceeds

oldest addr	# addr	% live	Age of addresses (in days)				
			<1	$1-5$	$5-10$	$10-30$	>30
38 d*	243	28%	36	71	28	79	29
41 d*	162	28%	23	29	27	44	39
42 d*	244	19%	25	45	29	95	50
42 d*	195	23%	23	40	23	64	45
43 d*	219	20%	66	57	23	50	23
103 d	4096	8%	722	645	236	819	1674
127 d	4096	8%	90	290	328	897	2491
271 d	4096	8%	750	693	356	809	1488
240 d	4096	6%	419	445	32	79	3121
373 d	4096	5%	9	14	1	216	3856

Table 1: Age and churn of addresses in `tried` for our nodes (marked with *) and donated peers files.

85% with just $f = 72\%$; in the worst case for the attacker, this requires only 3000 bots, or an infrastructure attack of $s = 20$ groups and $t = 256$ addresses per group (5120 addresses). The same attack ($f = 72\%$, $\tau_a = 27$ minutes) running for just 4 hours still has $> 55\%$ success probability. To put this in context, if 3000 bots joined today's network (with < 7200 public-IP nodes [4]) and honestly followed the peer-to-peer protocol, they could eclipse a victim with probability $\approx (\frac{3000}{7200+3000})^8 = 0.006\%$.

5 Measuring Live Bitcoin Nodes

We briefly consider how parameters affecting the success of our eclipse attacks look on "typical" bitcoin nodes. We thus instrumented five bitcoin nodes with public IPs that we ran (continuously, without restarting) for 43 days from 12/23/2014 to 2/4/2015. We also analyze several peers files that others donated to us on 2/15/2015. Note that there is evidence of wide variations in metrics for nodes of different ages and in different regions [46]; as such, our analysis (Section 3-4) and some of our experiments (Section 6) focus on the attacker's worst-case scenario, where tables are initially full of fresh addresses.

Number of connections. Our attack requires the victim to have available slots for incoming connections. Figure 6 shows the number of connections over time for one of our bitcoin nodes, broken out by connections to public or private IPs. There are plenty of available slots; while our node can accommodate 125 connections, we never see more than 60 at a time. Similar measurements in [17] indicate that 80% of bitcoin peers allow at least 40 incoming connections. Our node saw, on average, 9.9 connections to public IPs over the course of its lifetime; of these, 8 correspond to *outgoing* connections, which means we rarely see incoming connections from public IPs. Results for our other nodes are similar.

Connection length. Because public bitcoin nodes rarely drop outgoing connections to their peers (except upon restart, network failure, or due to blacklisting, see Section 2.3), many connections are fairly long lived. When we sampled our nodes on 2/4/2015, across all of our nodes, 17% of connections had lasted more than 15 days, and of these, 65.6% were to public IPs. On the other hand, many bitcoin nodes restart frequently; we saw that 43% of connections lasted less than two days and of these, 97% were to nodes with private IPs. This may explain why we see so few incoming connections from public IPs; many public-IP nodes stick to their mature long-term peers, rather than our young-ish nodes.

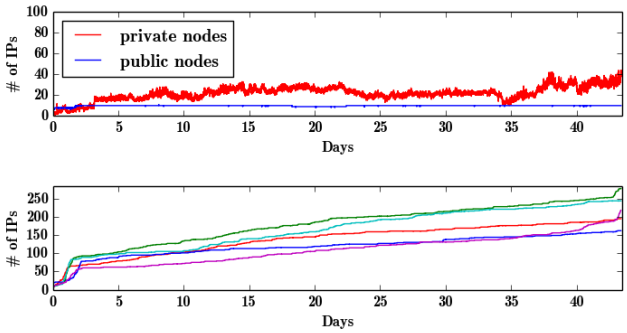


Figure 6: (Top) Incoming + outgoing connections vs time for one of our nodes. (Bottom) Number of addresses in `tried` vs time for all our nodes.

Size of `tried` and `new` tables. In our worst case attack, we supposed that the `tried` and `new` tables were completely full of fresh addresses. While our Bitcoin nodes' `new` tables filled up quite quickly (99% within 48 hours), Table 1 reveals that their `tried` tables were far from full of fresh addresses. Even after 43 days, the `tried` tables for our nodes were no more than $300/4096 \approx 8\%$ full. This likely follows because our nodes had very few incoming connections from public IPs; thus, most addresses in `tried` result from successful outgoing connections to public IPs (infrequently) drawn from `new`.

Freshness of `tried`. Even those few addresses in `tried` are not especially fresh. Table 1 shows the age distribution of the addresses in `tried` for our nodes and from donated peers files. For our nodes, 17% of addresses were more than 30 days old, and 48% were more than 10 days old; these addresses will therefore be less preferred than the adversarial ones inserted during an eclipse attack, even if the adversary does not invest much time τ_ℓ in attacking the victim.

Churn. Table 1 also shows that a small fraction of addresses in `tried` were online when we tried connecting to them on 2/17/2015.[4] This suggests further vulnerability to eclipse attacks, because if most legitimate addresses in `tried` are offline when a victim resets, the victim is likely to connect to an adversarial address.

[4]For consistency with the rest of this section, we tested our nodes tables from 2/4/2015. We also repeated this test for tables taken from our nodes on 2/17/2015, and the results did not deviate more than 6% from those of Table 1.

Attack Type	Attacker resources: grps s	addrs/ grp t	total addrs	τ_ℓ, time invest	τ_a, round	Experiment: Total pre-attack `new`	Total pre-attack `tried`	Total post-attack `new`	Total post-attack `tried`	Attack addrs `new`	Attack addrs `tried`	Wins	Predicted: Attack addrs `new`	Attack addrs `tried`	Wins
Infra (Worstcase)	32	256	8192	10 h	43 m	16384	4090	16384	4096	15871	3404	**98%**	16064	3501	**87%**
Infra (Transplant)	20	256	5120	1 hr	27 m	16380	278	16383	3087	14974	2947	**82%**	15040	2868	**77%**
Infra (Transplant)	20	256	5120	2 hr	27 m	16380	278	16383	3088	14920	2966	**78%**	15040	2868	**87%**
Infra (Transplant)	20	256	5120	4 hr	27 m	16380	278	16384	3088	14819	2972	**86%**	15040	2868	**91%**
Infra (Live)	20	256	5120	1 hr	27 m	16381	346	16384	3116	14341	2942	**84%**	15040	2868	**75%**
Bots (Worstcase)	2300	2	4600	5 h	26 m	16080	4093	16384	4096	16383	4015	**100%**	16384	4048	**96%**
Bots (Transplant)	200	1	200	1 hr	74 s	16380	278	16384	448	16375	200	**60%**	16384	200	**11%**
Bots (Transplant)	400	1	400	1 hr	90 s	16380	278	16384	648	16384	400	**88%**	16384	400	**34%**
Bots (Transplant)	400	1	400	4 hr	90 s	16380	278	16384	650	16383	400	**84%**	16384	400	**61%**
Bots (Transplant)	600	1	600	1 hr	209 s	16380	278	16384	848	16384	600	**96%**	16384	600	**47%**
Bots (Live)	400	1	400	1 hr	90 s	16380	298	16384	698	16384	400	**84%**	16384	400	**28%**

Table 2: Summary of our experiments.

6 Experiments

We now validate our analysis with experiments.

Methodology. In each of our experiments, the victim (bitcoind) node is on a virtual machine on the attacking machine; we also instrument the victim's code. The victim node runs on the public bitcoin network (*aka,* mainnet). The attacking machine can read all the victim's packets to/from the public bitcoin network, and can therefore forge TCP connections from arbitrary IP addresses. To launch the attack, the attacking machine forges TCP connections from each of its attacker addresses, making an incoming connection to the victim, sending a `VERSION` message and sometimes also an `ADDR` message (per Appendix B) and then disconnecting; the attack connections, which are launched at regular intervals, rarely occupy all of the victim's available slots for incoming connections. To avoid harming the public bitcoin network, (1) we use "reserved for future use" [43] IPs in 240.0.0.0/8-249.0.0.0/8 as attack addresses, and 252.0.0.0/8 as "trash" sent in `ADDR` messages, and (2) we drop any `ADDR` messages the (polluted) victim attempts to send to the public network.

At the end of the attack, we repeatedly restart the victim and see what outgoing connections it makes, dropping connections to the "trash" addresses and forging connections for the attacker addresses. If all 8 outgoing connections are to attacker addresses, the attack succeeds, and otherwise it fails. Each experiment restarts the victim 50 times, and reports the fraction of successes. At each restart, we revert the victim's tables to their state at the end of the attack, and rewind the victim's system time to the moment the attack ended (to avoid dating timestamps in `tried` and `new`). We restart the victim 50 times to measure the success rate of our (probabilistic) attack; in a real attack, the victim would only restart once.

Initial conditions. We try various initial conditions:

1. Worst case. In the attacker's worst-case scenario, the victim initially has `tried` and `new` tables that are completely full of legitimate addresses with fresh timestamps. To set up the initial condition, we run our attack for no longer than one hour on a freshly-born victim node, filling `tried` and `new` with IP addresses from 251.0.0.0/8, 253.0.0.0/8 and 254.0.0.0/8, which we designate as "legitimate addresses"; these addresses are no older than one hour when the attack starts. We then restart the victim and commence attacking it.

2. Transplant case. In our transplant experiments, we copied the `tried` and `new` tables from one of our five live bitcoin nodes on 8/2/2015, installed them in a fresh victim with a different public IP address, restarted the victim, waited for it to establish eight outgoing connections, and then commenced attacking. This allowed us to try various attacks with a consistent initial condition.

3. Live case. Finally, on 2/17/2015 and 2/18/2015 we attacked our live bitcoin nodes while they were connected to the public bitcoin network; at this point our nodes had been online for 52 or 53 days.

Results (Table 2). Results are in Table 2. The first five columns summarize attacker resources (the number of groups s, addresses per group t, time invested in the attack τ_ℓ, and length of a round τ_a per Sections 3-4). The next two columns present the initial condition: the number of addresses in `tried` and `new` prior to the attack. The following four columns give the size of `tried` and `new`, and the number of attacker addresses they store, at the end of the attack (when the victim first restarts). The *wins* columns counts the fraction of times our attack succeeds after restarting the victim 50 times.

The final three columns give predictions from Sections 3.3, 4. The *attack addrs* columns give the expected number of addresses in `new` (Appendix B) and `tried`. For `tried`, we assume that the attacker runs his attack for enough rounds so that the expected number of addresses in `tried` is governed by equation (4) for the botnet, and the 'initially empty' curve of Figure 4 for the infrastructure attack. The final column predicts success per Section 3.3 using *experimental values* of τ_a, τ_ℓ, f, f'.

Observations. Our results indicate the following:

1. Success in worst case. Our experiments confirm that an infrastructure attack with 32 groups of size /24 (8192

attack addresses (total) succeeds in the worst case with very high probability. We also confirm that botnets are superior to infrastructure attacks; 4600 bots had 100% success even with a worst-case initial condition.

2. Accuracy of predictions. Almost all of our attacks had an experimental success rate that was *higher* than the predicted success rate. To explain this, recall that our predictions from Section 3.3 assume that legitimate addresses are exactly τ_ℓ old (where τ_ℓ is the time invested in the attack); in practice, legitimate addresses are likely to be even older, especially when we work with `tried` tables of real nodes (Table 1). Thus, Section 3.3's predictions are a lower bound on the success rate.

Our experimental botnet attacks were dramatically more successful than their predictions (*e.g.,* 88% actual vs. 34% predicted), most likely because the addresses initially in `tried` were already very stale prior to the attack (Table 1). Our infrastructure attacks were also more successful then their predictions, but here the difference was much less dramatic. To explain this, we look to the `new` table. While our success-rate predictions assume that `new` is completely overwritten, our infrastructure attacks failed to completely overwrite the `new` table;[5] thus, we have some extra failures because the victim made outgoing connections to addresses in `new`.

3. Success in a 'typical' case. Our attacks are successful with even fewer addresses when we test them on our live nodes, or on tables taken from those live nodes. Most strikingly, a small botnet of 400 bots succeeds with very high probability; while this botnet completely overwrites `new`, it fills only $400/650 = 62\%$ of `tried`, and still manages to win with more than 80% probability.

7 Countermeasures

We have shown how an attacker with enough IP addresses and time can eclipse any target victim, regardless of the state of the victim's `tried` and `new` tables. We now present countermeasures that make eclipse attacks more difficult. Our countermeasures are inspired by botnet architectures (Section 8), and designed to be faithful to bitcoin's network architecture.

The following five countermeasures ensure that: (1) If the victim has h legitimate addresses in `tried` before the attack, and a p-fraction of them accept incoming connections during the attack when the victim restarts, then even an attacker *with an unbounded number of addresses* cannot eclipse the victim with probability exceeding equation (10). (2) If the victim's oldest outgoing connection is to a legitimate peer before the attack, then the eclipse attack *fails* if that peer accepts incoming connections when the victim restarts.

1. Deterministic random eviction. Replace bitcoin eviction as follows: just as each address deterministically hashes to a single bucket in `tried` and `new` (Section 2.2), an address also deterministically hashes to a single slot in that bucket. This way, an attacker cannot increase the number of addresses stored by repeatedly inserting the same address in multiple rounds (Section 4.1). Instead, addresses stored in `tried` are given by the 'random eviction' curves in Figures 2, 4, reducing the attack addresses stored in `tried`.

2. Random selection. Our attacks also exploit the heavy bias towards forming outgoing connections to addresses with fresh timestamps, so that an attacker that owns only a small fraction $f = 30\%$ of the victim's `tried` table can increase its success probability (to say 50%) by increasing τ_ℓ, the time it invests in the attack (Section 3.3). We can eliminate this advantage for the attacker if addresses are selected at random from `tried` and `new`; this way, a success rate of 50% always requires the adversary to fill $\sqrt[8]{0.5} = 91.7\%$ of `tried`, which requires 40 groups in an infrastructure attack, or about 3680 peers in a botnet attack. Combining this with deterministic random eviction, the figure jumps to 10194 bots for 50% success probability.

These countermeasures harden the network, but still allow an attacker with enough addresses to overwrite all of `tried`. The next countermeasure remedies this:

3. Test before evict. Before storing an address in its (deterministically-chosen) slot in a bucket in `tried`, first check if there is an older address stored in that slot. If so, briefly attempt to connect to the older address, and if connection is successful, then the older address is *not* evicted from the `tried` table; the new address is stored in `tried` only if the connection fails.

We analyze these three countermeasures. Suppose that there are h legitimate addresses in the `tried` table prior to the attack, and model network churn by supposing that each of the h legitimate addresses in `tried` is live (*i.e.,* accepts incoming connections) independently with probability p. With test-before-evict, the adversary cannot evict $p \times h$ legitimate addresses (in expectation) from `tried`, regardless of the number of distinct addresses it controls. Thus, even if the rest of `tried` is full of adversarial addresses, the probability of eclipsing the victim is bounded to about

$$\Pr[\text{eclipse}] = f^8 < \left(1 - \tfrac{p \times h}{64 \times 64}\right)^8 \qquad (10)$$

This is in stark contrast to today's protocol, where attackers with enough addresses have *unbounded* success probability even if `tried` is *full* of legitimate addresses.

[5] The `new` table holds 16384 addresses and from 6^{th} last column of Table 2 we see the `new` is not full for our infrastructure attacks. Indeed, we predict this in Appendix B.

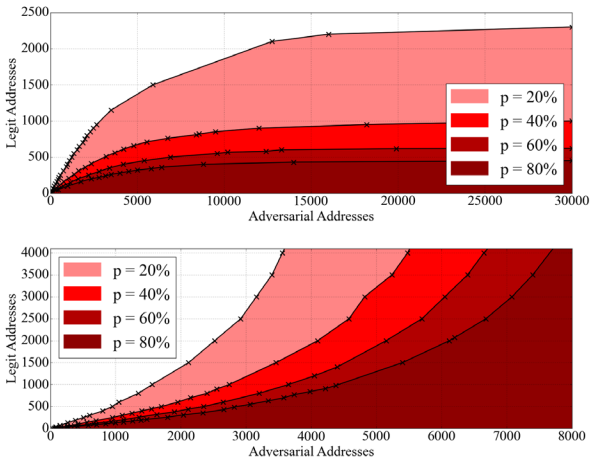


Figure 7: The area below each curve corresponds to a number of bots a that can eclipse a victim with probability at least 50%, given that the victim initially has h legitimate addresses in `tried`. We show one curve per churn rate p. (Top) With test before evict. (Bottom) Without.

We perform Monte-Carlo simulations assuming churn p, h legitimate addresses initially stored in `tried`, and a botnet inserting a addresses into `tried` via unsolicited incoming connections. The area below each curve in Figure 7 is the number of bots a that can eclipse a victim with probability at least 50%, given that there are initially h legitimate addresses in `tried`. With test-before-evict, the curves plateau horizontally at $h = 4096(1-\sqrt[8]{0.5})/p$; as long as h is greater than this quantity, even a botnet *with an infinite number of addresses* has success probability bounded by 50%. Importantly, the plateau is absent without test-before-evict; a botnet with enough addresses can eclipse a victim *regardless* of the number of legitimate addresses h initially in `tried`.

There is one problem, however. Our bitcoin nodes saw high churn rates (Table 1). With a $p = 28\%$ churn rate, for example, bounding the adversary's success probability to 10% requires about $h = 3700$ addresses in `tried`; our nodes had $h < 400$. Our next countermeasure thus adds more legitimate addresses to `tried`:

4. Feeler Connections. Add an outgoing connection that establish short-lived test connections to randomly-selected addresses in `new`. If connection succeeds, the address is evicted from `new` and inserted into `tried`; otherwise, the address is evicted from `new`.

Feeler connections clean trash out of `new` while increasing the number of fresh address in `tried` that are likely to be online when a node restarts. Our fifth countermeasure is orthogonal to those above:

5. Anchor connections. Inspired by Tor entry guard rotation rates [33], we add two connections that persist between restarts. Thus, we add an `anchor` table, recording addresses of current outgoing connections and the time of first connection to each address. Upon restart, the node dedicates two extra outgoing connections to the oldest `anchor` addresses that accept incoming connections. Now, in addition to defeating our other countermeasures, a successful attacker must also disrupt `anchor` connections; eclipse attacks fail if the victim connects to an `anchor` address not controlled by the attacker.

Apart from these five countermeasures, a few other ideas can raise the bar for eclipse attacks:

6. More buckets. Among the most obvious countermeasure is to increase the size of the `tried` and `new` tables. Suppose we doubled the number of buckets in the `tried` table. If we consider the infrastructure attack, the buckets filled by s groups jumps from $(1-e^{-\frac{4s}{64}})$ (per equation (9) to $(1-e^{-\frac{4s}{128}})$. Thus, an infrastructure attacker needs double the number of groups in order to expect to fill the same fraction of `tried`. Similarly, a botnet needs to double the number of bots. Importantly, however, this countermeasure is helpful only when `tried` already contains many legitimate addresses, so that attacker owns a smaller fraction of the addresses in `tried`. However, if `tried` is mostly empty (or contains mostly stale addresses for nodes that are no longer online), the attacker will still own a large fraction of the addresses in `tried`, even though the number of `tried` buckets has increased. Thus, this countermeasure should also be accompanied by another countermeasure (*e.g.,* feeler connections) that increases the number of legitimate addresses stored in `tried`.

7. More outgoing connections. Figure 6 indicates our test bitcoin nodes had at least 65 connections slots available, and [17] indicates that 80% of bitcoin peers allow at least 40 incoming connections. Thus, we can require nodes to make a few additional outgoing connections without risking that the network will run out of connection capacity. Indeed, recent measurements [51] indicate that certain nodes (*e.g.,* mining-pool gateways) do this already. For example, using twelve outgoing connections instead of eight (in addition to the feeler connection and two anchor connections), decreases the attack's success probability from f^8 to f^{12}; to achieve 50% success probability the infrastructure attacker now needs 46 groups, and the botnet needs 11796 bots.

8. Ban unsolicited `ADDR` messages. A node could choose not to accept large unsolicited `ADDR` messages (with > 10 addresses) from incoming peers, and only solicit `ADDR` messages from outgoing connections when its `new` table is too empty. This prevents adversarial incoming connections from flooding a victim's `new` table with trash addresses. We argue that this change is not harmful, since even in the current network, there is no shortage of address in the `new` table (Section 5). To make this more

concrete, note that a node request ADDR messages upon establishing an outgoing connection. The peer responds with n randomly selected addresses from its tried and new tables, where n is a random number between x and 2500 and x is 23% of the addresses the peer has stored. If each peer sends, say, about $n = 1700$ addresses, then new is already $8n/16384 = 83\%$ full the moment that the bitcoin node finishing establishing outgoing connections.

9. Diversify incoming connections. Today, a bitcoin node can have all of its incoming connections come from the same IP address, making it far too easy for a single computer to monopolize a victim's incoming connections during an eclipse attack or connection-starvation attack [32]. We suggest a node accept only a limited number of connections from the same IP address.

10. Anomaly detection. Our attack has several specific "signatures" that make it detectable including: (1) a flurry of short-lived incoming TCP connections from diverse IP addresses, that send (2) large ADDR messages (3) containing "trash" IP addresses. An attacker that suddenly connects a large number of nodes to the bitcoin network could also be detected, as could one that uses eclipsing per Section 1.1 to dramatically decrease the network's mining power. Thus, monitoring and anomaly detection systems that look for this behavior are also be useful; at the very least, they would force an eclipse attacker to attack at low rate, or to waste resources on overwriting new (instead of using "trash" IP addresses).

Status of our countermeasures. We disclosed our results to the bitcoin core developers in 02/2015. They deployed Countermeasures 1, 2, and 6 in the bitcoind v0.10.1 release, which now uses deterministic random eviction, random selection, and scales up the number of buckets in tried and new by a factor of four. To illustrate the efficacy of this, consider the worst-case scenario for the attacker where tried is completely full of legitimate addresses. We use Lemma A.1 to estimate the success rate of a botnet with t IP addresses as

$$\Pr[\text{Eclipse}] \approx \left(1 - \left(\tfrac{16383}{16384}\right)^t\right)^8 \qquad (11)$$

Plotting (11) in Figure 8, we see that this botnet requires 163K addresses for a 50% success rate, and 284K address for a 90% success rate. This is good news, but we caution that ensuring that tried is full of legitimate address is still a challenge (Section 5), especially since there may be fewer than 16384 public-IP nodes in the bitcoin network at a given time. Countermeasures 3 and 4 are designed to deal with this, and so we have also developed a patch with these two countermeasures; see [40] for our implementation and its documentation.

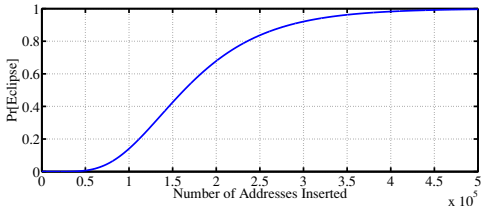


Figure 8: Probability of eclipsing a node vs the number of addresses (bots) t for bitcoind v0.10.1 (with Countermeasures 1,2 and 6) when tried is initially full of legitimate addresses per equation (11).

8 Related Work

The bitcoin peer-to-peer (p2p) network. Recent work considers how bitcoin's network can delay or prevent block propagation [31] or be used to deanonymize bitcoin users [16, 17, 48]. These works discuss aspects of bitcoin's networking protocol, with [16] providing an excellent description of ADDR message propagation; we focus instead on the structure of the tried and new tables, timestamps and their impact on address selection (Section 2). [17] shows that nodes connecting over Tor can be eclipsed by a Tor exit node that manipulates both bitcoin and Tor. Other work has mapped bitcoin peers to autonomous systems [38], geolocated peers and measured churn [34], and used side channels to learn the bitcoin network topology [16, 51].

p2p and botnet architectures. There has been extensive research on eclipse attacks [27, 61, 62] in structured p2p networks built upon distributed hash tables (DHTs); see [64] for a survey. Many proposals defend against eclipse attacks by adding more structure; [61] constrains peer degree, while others use constraints based on distance metrics like latency [42] or DHT identifiers [13]. Bitcoin, by contrast, uses an unstructured network. While we have focused on exploiting specific quirks in bitcoin's existing network, other works *e.g.,* [11, 15, 21, 44] design new unstructured networks that are robust to Byzantine attacks. [44] blacklists misbehaving peers. Puppetcast's [15] centralized solution is based on public-key infrastructure [15], which is not appropriate for bitcoin. Brahms [21] is fully decentralized, and instead constrains the rate at which peers exchange network information—a useful idea that is a significant departure from bitcoin's current approach. Meanwhile, our goals are also more modest than those in these works; rather than requiring that each node is *equally likely* to be sampled by an honest node, we just want to limit eclipse attacks on initially well-connected nodes. Thus, our countermeasures are inspired by botnet architectures, which share this same goal. Rossow *et al.* [59] finds that many botnets, like bitcoin, use unstructured peer-to-peer networks and gossip (*i.e.,* ADDR messages), and describes

how botnets defend against attacks that flood local address tables with bogus information. The Sality botnet refuses to evict "high-reputation" addresses; our `anchor` countermeasure is similar (Section 7). Storm uses test-before-evict [30], which we have also recommended for bitcoin. Zeus [12] disallows connections from multiple IP in the same /20, and regularly clean tables by testing if peers are online; our feeler connections are similar.

9 Conclusion

We presented an eclipse attack on bitcoin's peer-to-peer network that undermines bitcoin's core security guarantees, allowing attacks on the mining and consensus system, including *N*-confirmation double spending and adversarial forks in the blockchain. Our attack is for nodes with public IPs. We developed mathematical models of our attack, and validated them with Monte Carlo simulations, measurements and experiments. We demonstrated the practically of our attack by performing it on our own live bitcoin nodes, finding that an attacker with 32 distinct /24 IP address blocks, or a 4600-node botnet, can eclipse a victim with over 85% probability in the attacker's *worst case*. Moreover, even a 400-node botnet sufficed to attack our own live bitcoin nodes. Finally, we proposed countermeasures that make eclipse attacks more difficult while still preserving bitcoin's openness and decentralization; several of these were incorporated in a recent bitcoin software upgrade.

Acknowledgements

We thank Foteini Baldimtsi, Wil Koch, and the USENIX Security reviewers for comments on this paper, various bitcoin users for donating their peers files, and the bitcoin core devs for discussions and for implementing Countermeasures 1,2,6. E.H., A.K., S.G. were supported in part by NSF award 1350733, and A.Z. by ISF Grants 616/13, 1773/13, and the Israel Smart Grid (ISG) Consortium.

References

[1] Amazon web services elastic ip. `http://aws.amazon.com/ec2/faqs/#elastic-ip`. Accessed: 2014-06-18.

[2] Bitcoin: Common vulnerabilities and exposures. `https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures`. Accessed: 2014-02-11.

[3] Bitcoin wiki: Double-spending. `https://en.bitcoin.it/wiki/Double-spending`. Accessed: 2014-02-09.

[4] Bitnode.io snapshot of reachable nodes. `https://getaddr.bitnodes.io/nodes/`. Accessed: 2014-02-11.

[5] Bitpay: What is transaction speed? `https://support.bitpay.com/hc/en-us/articles/202943915-What-is-Transaction-Speed-`. Accessed: 2014-02-09.

[6] Bug bounty requested: 10 btc for huge dos bug in all current bitcoin clients. Bitcoin Forum. `https://bitcointalk.org/index.php?topic=944369.msg10376763#msg10376763`. Accessed: 2014-06-17.

[7] CVE-2013-5700: Remote p2p crash via bloom filters. `https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures`. Accessed: 2014-02-11.

[8] Microsoft azure ip address pricing. `http://azure.microsoft.com/en-us/pricing/details/ip-addresses/`. Accessed: 2014-06-18.

[9] Rackspace: Requesting additional ipv4 addresses for cloud servers. `http://www.rackspace.com/knowledge_center/article/requesting-additional-ipv4-addresses-for-cloud-servers`. Accessed: 2014-06-18.

[10] Ghash.io and double-spending against betcoin dice, October 30 2013.

[11] ANCEAUME, E., BUSNEL, Y., AND GAMBS, S. On the power of the adversary to solve the node sampling problem. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XI*. Springer, 2013, pp. 102–126.

[12] ANDRIESSE, D., AND BOS, H. An analysis of the zeus peer-to-peer protocol, April 2014.

[13] AWERBUCH, B., AND SCHEIDELER, C. Robust random number generation for peer-to-peer systems. In *Principles of Distributed Systems*. Springer, 2006, pp. 275–289.

[14] BAHACK, L. Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013* (2013).

[15] BAKKER, A., AND VAN STEEN, M. Puppetcast: A secure peer sampling protocol. In *European Conference on Computer Network Defense (EC2ND)* (2008), IEEE, pp. 3–10.

[16] BIRYUKOV, A., KHOVRATOVICH, D., AND PUSTOGAROV, I. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 15–29.

[17] BIRYUKOV, A., AND PUSTOGAROV, I. Bitcoin over tor isn't a good idea. *arXiv preprint arXiv:1410.6079* (2014).

[18] BITCOIN WIKI. Confirmation. `https://en.bitcoin.it/wiki/Confirmation`, February 2015.

[19] BITCOIN WISDOM. Bitcoin difficulty and hash rate chart. `https://bitcoinwisdom.com/bitcoin/difficulty`, February 2015.

[20] BLOCKCHAIN.IO. Average transaction confirmation time. `https://blockchain.info/charts/avg-confirmation-time`, February 2015.

[21] BORTNIKOV, E., GUREVICH, M., KEIDAR, I., KLIOT, G., AND SHRAER, A. Brahms: Byzantine resilient random membership sampling. *Computer Networks 53*, 13 (2009), 2340–2359.

[22] BRANDS, S. Untraceable off-line cash in wallets with observers (extended abstract). In *CRYPTO* (1993).

[23] CAIDA. AS to Organization Mapping Dataset, July 2014.

[24] CAIDA. Routeviews prefix to AS Mappings Dataset for IPv4 and IPv6, July 2014.

[25] CAMENISCH, J., HOHENBERGER, S., AND LYSYANSKAYA, A. Compact e-cash. In *EUROCRYPT* (2005).

[26] CARNABOTNET. Internet census 2012. `http://internetcensus2012.bitbucket.org/paper.html`, 2012.

Peer-to-Peer Networking and Applications (2019) 12:951–968
https://doi.org/10.1007/s12083-018-0701-x

Reputation based approach for improved fairness and robustness in P2P protocols

Francis N. Nwebonyi[1] · Rolando Martins[1] · Manuel E. Correia[1]

Received: 8 November 2017 / Accepted: 19 November 2018 / Published online: 6 December 2018

Abstract
Peer-to-Peer (P2P) overlay networks have gained popularity due to their robustness, cost advantage, network efficiency and openness. Unfortunately, the same properties that foster their success, also make them prone to several attacks. To mitigate these attacks, several scalable security mechanisms which are based on the concepts of trust and reputation have been proposed. These proposed methods tend to ignore some core practical requirements that are essential to make them more useful in the real world. Some of such requirements include efficient bootstrapping of each newcomer's reputation, and mitigating seeder(s) exploitation. Additionally, although interaction among participating peers is usually the bases for reputation, the importance given to the frequency of interaction between the peers is often minimized or ignored. This can result in situations where barely known peers end-up having similar trust scores to the well-known and consistently cooperative nodes. After a careful review of the literature, this work proposes a novel and scalable reputation based security mechanism that addresses the aforementioned problems. The new method offers more efficient reputation bootstrapping, mitigation of bandwidth attack and better management of interaction rate, which further leads to improved fairness. To evaluate its performance, the new reputation model has been implemented as an extension of the BitTorrent protocol. Its robustness was tested by exposing it to popular malicious behaviors in a series of extensive PeerSim simulations. Results show that the proposed method is very robust and can efficiently mitigate popular attacks on P2P overlay networks.

Keywords Trust · P2P · Edge clouds · Reputation · And security

1 Introduction

Computing is currently drifting towards the network edge due to increased interest in decentralized solutions that offer enhanced privacy preserving algorithms, better data locality and alternative communication infrastructures. As an example, there are growing opportunities to engage the resources and properties of mobile devices in creating platforms to harvest resources locally, through the use of hyper local clouds, also called edge clouds [1]. These edge clouds offer a way to reduce dependency on centralized infrastructure (including standard cloud computing infrastructures), especially in highly dense environments such as stadiums, concerts and museums where communication infrastructures are normally under strain.

Generally speaking, P2P distributed networks are able to incorporate remote users by allowing them to connect easily and in an open manner to each other, thereby making communication cheaper, convenient and readily available. They can be very dynamic: self-organizing, self-discovering and self-adapting. This opposes the traditional centralized systems which can be expensive and sometimes lack availability. But several security issues remain unsolved [2], and different forms of aggressive attacks still target online P2P systems [3, 4].

The literature emphasizes trust and reputation based methods as the way forward. This is demonstrated by many trust models which have already been proposed [5–11], targeting different P2P platforms. Successful online commercial services such as Ebay use some of these methods to help users to discern who to transact with, based on their reputation. Kazaa also used a reputation management system, where users rate the participation level of each other, by giving more grade to those that share

✉ Francis N. Nwebonyi
fnwebonyi@dcc.fc.up.pt

Rolando Martins
rmartins@dcc.fc.up.pt

Manuel E. Correia
mcc@dcc.fc.up.pt

[1] Faculty of Sciences, University of Porto, & CRACS/INESC-TEC, Porto, Portugal

genuine contents frequently [8]. Similarly, BitTorrent adopts a 'give and take' method known as tit-for-tat. Although this is not exactly a reputation based system, it also rewards users that contribute more to the network.

In this work, we have reviewed some popular trust and reputation models in the literature that are aimed at addressing security challenges in distributed networks and P2P in particular. By identifying and building on their strengths, we derived a method that can address their perceived weaknesses. Based on the review which is discussed in section 4, some of the noticed weaknesses (which also highlights our areas of key contributions) are summarized as follows:

I. Most earlier methods focus mainly on how to mitigate attacks from malicious nodes after they have joined the network, but pay less attention to stopping or limiting malicious peers from joining initially. Those who attempt to overcome this limitation often require pre-established trust relationships over other channels, or some central entity such as server or super nodes which can constitute prime target for attacks. The new method overcomes these barriers and provides a distributed and fairer means of bootstrapping reputation scores for newcomers.
II. Current methods tend to channel huge effort towards protecting client nodes from malicious server nodes (or seeders), but little attention is often given to the protection of server nodes from malicious clients, especially in distributed P2P platforms. This gives rise to a form of bandwidth attack which has escaped the attention of most trust and reputation models in the literature. This attack has been shown to have adverse potential effect on P2P networks such as BitTorrent [12, 13], it is further discussed in section 3. To the best of our knowledge, our work is the first to tackle bandwidth attack (on seeders) in a distributed P2P reputation model, based on actual leecher-to-leecher behavior of peers. The new method empowers both client and server nodes to directly identify attackers and act fairly.
III. It is a common practice to use interaction experience between "trustor" and "trustee" to determine reputation and trust. In some cases, number of good interactions minus the bad ones normally form the bases for direct reputation and trust scores. Other times, it is based on the ratio of good to total number of interactions. But the cumulative interaction rate barely reflects in the end result. This sometimes leads to situations where more familiar node(s) may have little or no noticeable advantage over a less familiar one. In the new method, this silent interaction rate concept (or 'familiarity') is accounted for, as a way of encouraging consistency among nodes. When nodes have similar reputation scores, the new method would give priority to the agent with higher interaction rate, to reward consistency. Each peer is also monitored to see how much they have invested in the network relative to their capabilities and with respect to how much they have gained from it. This concept played a significant role towards mitigating bandwidth attacks. It is further discussed in section 5.2.

In order to assess our approach, we tested it on a classical BitTorrent protocol, via detailed simulations. BitTorrent is used mainly because of its popularity and well-known behavior, but our approach is designed to be adaptable to other similar P2P protocols. Our implementation has been done using PeerSim, an event based simulator [14] which has an implemented version of BitTorrent. The remaining part of this work is organized as follows; an introduction to trust concepts is given in section 2. Section 3, provides an overview of BitTorrent, and describes some well-known attacks on P2P networks, while section 4 discusses some related works. The proposed method is presented in section 5, succeeded by detailed description of experiments and discussion of results in section 6. Conclusion and future work then appears in section 7.

2 Trust concept

Trust has been identified as a crucial concept in digital security, without which it would be difficult to reason about the security of any system in a convincing manner [15]. While it is difficult to narrow trust down to a single universal definition, we adopt the view of Gambetta et al. [16] who defined it as a level of subjective probability on which a party bases his assessment that another party will act in a certain manner, before monitoring the action (or even if he is not able to monitor it), and given that the assessment affects his own action.

To consider an entity trustworthy means to have high probability that such entity will act favorably or at least in a manner that is not harmful, within a given context. Trust is context dependent, more like saying that you can trust a lawyer's legal advice but not his advice on how to fix your car. It is usually based on some sort of relationship; direct, indirect or both.

Reputation is usually a basis for determining trust, which implies that if an entity has maintained a good reputation during past interactions, it can be accorded a level of trust that would warrant subsequent interaction, within a context. Reputation can be seen as a view about an entity within a community, which is generally known and assessed by the members of that community [17].

Broadly speaking, trust can be direct, indirect or hybrid. Direct trust is achieved by observing a neighbor directly, without involving a third party. On the other hand, the concept of trust transitivity is usually harnessed to establish indirect trust, which involves gathering recommendations from neighbors about an entity. The indirect trust of a trustee is dependent on the observation of other entities concerning its behavior during their past interactions. These observations are usually

communicated to the trustor in the form of recommendations. A trustor is a node who decides whether or not to trust another peer, the trustee.

There is also a hybrid of these two forms of trust. It involves the combination of direct and indirect trust to achieve even a stronger paradigm. This becomes particularly important when the trustor does not have enough information based on direct experience, to make decision concerning the trustee.

Security has been broadly classified into two categories; hard security and soft security [18]. Hard security is associated with traditional security approaches such as traditional access control and authentication. It provides partial security, because only corporate resources are protected from unauthorized users, but it does not always protect legitimate users from malicious service providers. This means that a central entity (usually the service provider) is able to assess peers that are requesting service from him, but the service requester nodes are not empowered to assess such service provider in a distributed manner. Soft security tackles this weakness, and trust is often promoted as the basis of soft security [19].

2.1 Reputation computation

For nodes to communicate on trust platforms, it is necessary for them to be able to measure the level of trust they can place on each other. This influences the disposition of the nodes to engage in transactions, or otherwise.

It can be challenging to measure digital trust with absolute precision, especially in distributed environments. The fact that many peers posses limited resources and capacities, adds to this challenge. Moreover, trust is a social concept, and some human instincts and reasoning which aid people in making trust decisions, are not easy to capture in algorithmic form. However, the literature contains series of approaches which have been proposed for the purpose of capturing, measuring and computing trust for digital environments [17]. Most of these proposals adopt one (or more) of the following methods to calculate reputations, upon which trust is based.

2.1.1 Summation

The simplest form of trust computation is usually to sum the positive and negative scores separately, then subtracting the negative from the positive, in order to determine how reliable an agent has been. This approach has been adopted to calculate direct reputation in [5], and it is also used by ebay reputation system [20].

2.1.2 Average

Another method which is close to simple summation involves taking the average of all scores of the peer being assessed, such as the case of Amazon [21]. Some other models such as TE-AODV [22] also use this method, assigning default score to newcomers. Sometimes, this method is slightly extended, by taking the weighted average, instead of the normal average. The extended version allows for taking the reputation of the recommending node (or similar factors) into account, as done in [8].

2.1.3 Bayesian Method

Interaction outputs are usually rated as good or bad, that is, in a binary form. The number of positive and negative scores are used to compute reputation, using the beta probability density function (PDF). The PDF tuple α and β represent the number of good and bad interactions respectively. The probability expectation value of beta distribution is represented as follows [23]:

$$E(p) = \alpha/(\alpha + \beta). \tag{1}$$

Compared to other methods, this method tend to have detailed theoretical background which can amount to higher confidence on its outcome. During bootstrapping, α and β are assigned the value of 1 each, which amounts to a default score of 0.5. Despite being popular, this type of (default) score initialization has been criticized for having the likelihood of being either unfair to the new node or the ones already in the network. It also has a tendency of not adequately addressing white washing [24, 25].

2.1.4 Fuzzy Techniques

This is one of the methods adopted for computing and aggregating reputation [8]. It involves reasoning about trust in-terms of fuzzy values. It appears to give more room for incorporating human-like reasoning such as agreement compliance, transaction time/age, etc., in reputation computation and aggregation. Given many recommendations for example, fuzzy technique can be applied to probe for the relevance of such recommendations by reasoning about compliance (or otherwise) to transaction agreement. That is, applying compliance as an objective measure to verify the subjective user ratings [26]. Different techniques apply various means to capture trust, but fuzzy techniques appear popular in this kind of reasoning.

2.1.5 Flow Techniques

Here, reputation has to do with the level of importance that a peer attracts within a community. As an example, in PageRank [27], the number of links pointing to an entity (such as website) compared to the number of links that leaves the entity, is used to determine the importance or reputation of that entity. If we take I(P) to be the set of other entities that are pointing to page P, otherwise called the in-links, and $|P|$ to be the number of other pages that P is pointing out to, otherwise

called the outlink. Then PageRank of P denoted by r(P), is as follows:

$$r(P) = \sum_{Q \in I(P)} \frac{r(Q)}{|Q|}, \quad (2)$$

where $|Q|$ is the out-links from Q. r(P) is mainly influenced by the pages that are pointing to it, which can be regarded as recommendations. However, an increase in the number of references made by Q, reduces the influence of Q on P. As the inlinks of Q grow, the rank or reputation of P also grows.

2.1.6 Belief Models

This approach is based on probability theory, but the sum of probabilities for all possible outcomes do not necessarily have to be equal to 1. Trust is expressed as a belief that a system will resist malicious attacks, or that a person will cooperate and not defect. Unlike some other models in which belief in an agent's cooperation is considered true or false (i.e. discrete), belief models consider a case where there is no clear information on either belief or disbelief, and thus an uncertainty factor is introduced. This is done in a way that instead of using belief or disbelief in order to determine trust, uncertainty also counts. Opinions are usually expressed in the form [28]: $\omega_Y^X = (b, d, u, a)$, where ω_Y^X is X's opinion about statement Y, while b, d, and u stand for belief, disbelief, and uncertainty receptively. a∈[0, 1] is referred to as relative atomicity, which is used to determine how uncertainty affects the expected opinion.

2.1.7 Discrete Trust

Trust is a social concept and thus humans as social entities are often better at determining the trust of other entities, based on experience and other intangible factors. Computationally, scholars try to capture as much of the human related attributes as they can, so as to make algorithmic trust possible and close to the human perception of trust. For example, some models such as [17] have emulated the human style of trust perception, to discreetly rank trust as very trustworthy, trustworthy, untrustworthy, and very untrustworthy. Lookup tables are used to then determine the actual trust of an entity.

3 BitTorrent

BitTorrent is the most popular among P2P protocols, accounting for about 53% of the entire P2P traffic, and more than 30% of overall Internet traffic [29]. It is a good case study because it is well known, widely deployed and it harbors major properties and behaviors of P2P systems, with some added peculiar features.

A share process in BitTorrent is initiated by first creating a meta-data describing the file that is to be shared, including SHA1 hash information about the pieces, among others. The information in the torrent file guides peers to the tracker and subsequently to other leechers and seeders in the swarm [30]. Leechers are peers that do not have all the file pieces, while seeders are peers with all the pieces. A tit-for-tat method is used to ensure that leechers who receive files also give to others. Seeders are assets to the network because they selflessly share files, without the need to download. Because of this, the tit- for-tat does not directly apply to them and thus they can be exploited in some ways, as discussed in the following subsection.

In DHT (Distributed Hash Table) based BitTorrent, the tracker does not aid in the discovery of files or peerset. Nodes do this all by themselves. With distributed tracker systems such as Mainline DHT (MLDHT), peers use content IDs (also called infohash) to find the location of desired contents. Once a node knows the infohash of the file it wishes to fetch, it applies some controls to arrive at its location. Some of the controls include; 'PING' for ascertaining node's availability, 'FIND NODE' used to get the k closest neighbors, 'GET PEERS' for getting the initial peerset and 'ANNOUNCE PEER' used by nodes to announce that they are part of the swarm [31]. In our method, the tracker also does not play any special role; it is focused on individual peers and thus can function without any central agent.

3.1 P2P attacks

Decentralized networks suffer from diverse forms of attack. Apart from free-riding, which is an act of selfishness, in which peers download from others without uploading to them in return, some other popular attacks in BitTorrent and P2P include the following:

1. *Lying Piece Attack:* The goal of this attack is to destabilize the rarest first policy of Bittorrent, in which peers while downloading, give priority to blocks that are less available in the swarm, so that they will not be lost completely. Attackers advertise false pieces, thereby misleading other peers on the pieces that are actually rare [4]. More generally, the intention of attackers here might also be to cause other forms of confusion, such as diverting the attention of victims from genuine contents to non-existing ones, in order to frustrate their download efforts.
2. *Chatty Peer Attack:* Attackers establish many TCP connections with the victims. They announce possession of many file pieces, but when the victims request for some blocks of such pieces, they never upload any. Instead, the attackers resend handshake messages, thereby sticking as

neighbors to the victims, who spend considerable amount of time waiting in vain for the attacker's response [32]. This can be an extension of the previous attack.

3. *Fake-Block Attack:* Attackers also advertise possession of many or all blocks. When victims request for such blocks, they send fake blocks in response. After downloading all blocks of a piece (from attackers and genuine peers), the victim checks if they are genuine. This check fails because of the fake blocks, warranting that the entire piece be downloaded afresh, meaning a huge waste of bandwidth and time [3].
4. *Bandwidth Attack:* This is usually an attack targeted at the seeders with the goal of occupying their upload bandwidth, so that there is little or none left for legitimate peers. Specifically, BitTorrent seeders do not download, so they keep uploading to the fastest downloading peer(s). Attackers therefore exploit this by simply connecting to seeders and downloading only from them at fastest rate possible, so that they are constantly unchoked, while the reputable peers are continually choked [12, 13, 33]. A version of this attack can also apply to other P2P networks, since there are always nodes that serve as resource donors (seeders) in the network. Attackers can target those server nodes in various ways, thereby obstructing their services.
5. *Sybil Attack:* Fake identities, otherwise called sybils can be created very cheaply. To beat reputation systems, attackers create links between its sybils, and through that means unleash different kinds of attack, e.g. bad-mouth attack. They can also use such links for raising their own reputation and thus gaining unmerited advantage [34, 35].
6. *Index Poisoning Attack:* In P2P file sharing platforms such as BitTorrent, clients keep indexes of files which link each file identifier to their hosts. Index poisoning attackers exploit this setting by publishing fake file indexes, with the goal of stopping their victims from accessing genuine neighbors. The attacker basically starts a file sharing task but with fake information such as port numbers, IP addresses, infohash, etc. Since there are no strong verification means, owing to the openness of the system, this attack is easily executed. The innocent peers are left to pay the price by using up their time and resources trying to access contents that are not existing; leading to denial of service [36].
7. *Combination Attack:* This is a form of content pollution attack, which combines fake-block and index poisoning attacks, to make a stronger impact. Attackers in this case extend index poisoning by including their own IDs in the fake infohash to be advertised. The idea is that if the victims insist on establishing connections despite failures due to fake information, they would eventually connect to the attacker(s) who make situation more difficult for them, by feeding them with fake blocks. So, they suffer the effects of both fake block attack and index poisoning attack. Combination attack has been illustrated to impose more harm than fake block or index poisoning attack individually [37].
8. *Peer Exchange (PEX) Attack:* This is very similar to distributed denial of service (DDoS) and index poisoning attacks. It is sometimes captured separately to portray the fact that they are launched on PEX; taking advantage of its features such as large list of peers (up to 3000 in some cases), and very frequent PEX massages. Although this is not the ideal design of PEX, studies have shown that almost all PEX implementations do not follow the original rules and there is no effective way of enforcing it [38]. Making it easy for malicious peers to fabricate huge PEX peer lists with false IP addresses, so victims waste resource and time trying in vain to connect. Similarly, the attacker can also send PEX massages containing the address and port number of the victim to as many others as it can. This is aimed at weighing the victims down with too many connection requests, resulting to DDoS.
9. *Collusion Attack:* Collusion attackers collude to favor themselves at the expense of legitimate peers. Attackers can join forces under collusion attack to make any of the mentioned attacks more devastating [39]. In P2P trust and reputation models, collusion attackers give themselves favorable scores, and downgrade the scores of others. This can mislead genuine peers into erroneously believing that the malicious nodes are the reputable ones, and vice versa. With this trick, attackers can take over the network and successfully eliminate the genuine nodes from network services.

4 Related work

Attempts have been made to mitigate attacks and unfairness in P2P, such as [3, 5, 40, 41]; each with their strengths and weaknesses. In this section, we review some of these reports, beginning with those that are specifically focused on BitTorrent, before exploring P2P in general. We intend to capture popular opinions in the field, and how they have addressed the focus of the proposed method.

A recent work [5] targets free-riding and similar attacks in BitTorrent. Most decisions are made by the tracker, such as decisions about the peers to service at any given time. This makes the method too centralized. The authors of [3] proposed a less centralized method in which 10% of the nodes act as super-nodes, and are charged with the responsibility of decision making. However, there is a possibility of some malicious nodes actually becoming super nodes, and then ruining the network. Moreover, this method also relies on central agents.

Wong et al. [7] mainly focused on content pollution. For any pollution suspected, the network is divided into sub units, in order to identify and carve out the pollutant. This technique can cause huge overhead and possible isolation of network parts. In the same vein, Santos et al. [42] proposed a reputation voting system based on subjective logic, where peers are required to vote for files as either polluted or whole. However, vote collection can take time, since the nodes might need to download big chunk of the file before casting their votes.

Sybil attack was the focus of [43] in Kademlia based BitTorrent. Nodes assess their potential service providers based on trust scores. In particular, trust scores are taken into account when ordering the k-bucket peers. New comers are kick started with some positive risk score to enable them join the network.

As noted earlier, these methods reflect the trend in most P2P reputation and trust models, which involves placing emphases mainly on client nodes, in order to shield them from malicious server nodes, but with minimal emphases on protecting the server nodes (seeders) from greedy leechers. This creates room for bandwidth attack, which has received little or no attention in current trust and reputation models. Some methods which are not reputation or trust based have attempted to address the challenge, but suffer crucial limitations related to poor or nonexistent means of verifying claims/votes [33], and outrageous overhead due to frequent encryption and decryption operations [44].

Another trend that is easily noticeable is that most methods bootstrap trust by assigning initial trust (or risk) score which does not reflect the newcomer's behavior, and which may be unfair to either the existing nodes or the newcomer. Some exceptions are subsequently discussed.

Besides BitTorrent, trust models for other P2P protocols, also portray similar trends. For instance, [8, 10] both initialize new comers with scores that allow them to join the network without initial assessment, thereby making white-washing very cheap. The proposals in [45, 46] require bootstrapping servers for new nodes to join, which can be a bottleneck in distributed settings.

Other methods such as [47] require some form of 'pre-established' trust relationship for newcomers, which may not always be available. An alternative solution which is based on Opennet model allows publicly known 'seednodes' to assist in introducing new nodes. However, this might involve revealing vital information to malicious nodes too, which they can use to attack the network. Additionally, if for any reason a 'seednode' becomes malicious, its effect would be high because it has high privilege. A Sybil resistant method presented in [48] is also based on pre-established trust relationship, which can be unavailable in some cases. Such relationships are expected to have been acquired offline, and are accumulated to form a bootstrap graph, which is a core part of their modified DHT routing mechanism.

In general, proposals that have targeted trust or reputation bootstrapping problem can be categorized into five [49]. The first and most popular method involves assigning default values to newcomers, examples of models belonging to this category were earlier discussed. The second is the dynamic initialization method [50], in which the current level of security within the network (or system) determines the scores that would be assigned to the new nodes. If maliciousness is currently high, then the value of initial trust score will be low, otherwise it would be high.

Thirdly, there is the recommendation and endorsement based method. Under this category, if a newcomer have similar interest and capabilities [24] or patterns [25], with an existing service (or node) which is already known as credible, then its initial score can be derived from such node or service. The fourth involves the use of game theory [51] for predicting initial trust, usually in web service platforms. While the fifth involves the use of central/super entities [52], or prior registration [49], or pre-existing trust relationship which was mentioned earlier [47, 48].

Another method involves the use of challenge/puzzle [53, 54] to restrict the number of fake identities that an attacker can introduce into the network. This is less trust-based and more related to resource testing. Since existing nodes need to verify the puzzle in order to introduce a newcomer, they may be placed at a disadvantage if the would-be attacker has more resources. Our method is different because the existing nodes do not have such disadvantage. The new method is also distributed, with no need for prior registration, pre-existing trust relationship or super node. It also addressed bandwidth attack which has been omitted in earlier systems.

The discussed related works are summarized in Tables 1 and 2. In Table 2, 'Y' stands for 'yes', '-' represents 'no', 'NA' means 'Not Applicable', while 'P' is for 'partial'. Our judgment for Sybil attack on the table agrees with an earlier work in [34], and more recently [43] which noted that no effective Sybil attack measure has been derived for distributed networks. We used the term 'partial' to mean that the reviewed method can minimize the attack, but not stop it completely. While 'yes' indicates those we think have addressed them more intensely. For want of space, we have not included methods that focus only on Bootstrapping in the tables, they have however been discussed previously.

5 Proposed method

The proposed approach is termed FBit (Fairer BitTorrent), it is aimed at achieving better fairness and robustness against maliciousness in P2P systems such as BitTorrent, with focus on bandwidth attack, fake block attack, Sybil attack and collusion attack. FBit ensures that every peer is treated with priority that reflects its level of cooperation, thereby providing better motivation for cooperation, better resource management, network

Table 1 Summary of related work

Ref.	Strength	Weakness
[9]	Copyright protection.	Pre-trusted and central entities.
[8]	Effective aggregation.	Resource intensive due to broadcasts.
[11]	Partial anonymity.	Not accountable for peers that drop requests.
[10]	Scores reflect global view.	Pre-trusted peers has to be trusted always.
[5]	Hard on free-riders.	Poor score sieve.
[3]	Scorer's reputation accessed.	Rogue nodes might make it to super level.
[33]	Independent decision	Fake votes easy.
[40]	Vote falsification is hard.	Resource intensive.
[42]	Early assessment.	Not DHT compatible.
[41]	Automatic scoring.	Only implicit, not established.
[7]	Checks fake blocks.	Tracker could be overworked.
[6]	Detects fake piece.	False positive/negative.
[43]	Distributed.	Only manages the effect of sybils.
[47]	Trust bootstrapping.	When pre-relationship is unavailable, the alternative can be weak.
[48]	Safe routing.	New comers require pre-trusted peers.
FBit	Distributed, efficient bootstrapping, bandwidth attack mitigation.	Churn effect not analyzed.

safety, fairness and efficiency. Although BitTorrent has been used for illustration here, the new method can be applied to similar P2P systems. The following subsections contain detailed explanation of the various components of the proposed method.

5.1 Reputation bootstrapping

At the point of entry, every node is expected to generate a key pair (*Kp*, *Kb*) with which they sign their transactions. New nodes start by discovering other nodes that are already in the swarm through the tracker (or DHT), in a usual BitTorrent pattern. Afterwards they send "Bitfield" (or similar control) massages requesting to be added as neighbors. Through the message, they also send their self-generated public keys and self-signed certificates to show that the request originated from them. When a trustor receives such request, it associates the accompanying public key with the trustee, to distinguish it from other nodes. For the newcomer's request to be granted, it has to demonstrate some preliminary trustworthy acts. Node's ID can be derived from its public key.

Table 2 More on summary of related work

Ref.	Bootstrapping Assessment	Fully Distributed	Content Pollution	Selfish Acts	Sybil Attack	Bandwidth Attack
[9]	–	–	Y	–	P	–
[8]	–	Y	Y	Y	P	–
[11]	–	Y	Y	Y	P	–
[10]	–	Y	Y	Y	P	–
[5]	–	–	Y	Y	P	–
[3]	–	–	Y	Y	P	–
[33]	NA	Y	–	–	–	Y
[40]	–	Y	Y	Y	P	–
[42]	–	–	Y	P	P	–
[41]	–	–	Y	P	P	–
[7]	–	–	Y	Y	–	–
[6]	–	Y	Y	Y	P	–
[43]	–	Y	Y	Y	Y	–
[47]	Y	Y	Y	Y	Y	–
[48]	Y	Y	–	Y	Y	–
FBit	Y	Y	Y	Y	Y	Y

As an illustration, assuming that *nodeA*, *nodeC* and *nodeE* were already in the network, when *nodeB* requests to join through *nodeA*. After *nodeB's* request, when *nodeA* receives the next "unchoke" message from any of its neighbors (e.g. *nodeC*), it responds by sending a request for a block, according to BitTorrent procedure. However, in addition, it requests that the block should be routed through *nodeB* (which is the new node), instead of having it sent directly. When *nodeB* receives such block from *nodeC*, it forwards it to *nodeA*, appending its signature, based on the key it generated initially.

Given that the transaction is honest, *nodeB* gains some reputation point, and the reputation of *nodeC* is updated too. This ensures that the new node does not gain reputation advantage over the existing ones. *NodeA* can repeat this step for some other blocks, giving *nodeB* more opportunities to serve and build reputation. In case of DHT, proximity would be considered in the routing process, such that nearby nodes would be more active in helping a newcomer to gain reputation.

While *nodeA* is waiting, it can proceed with other requests so that it is not trapped if *nodeB* does not deliver. Other nodes in the network such as *nodeC* and *nodeE* can also route some of their requests through *nodeB* in order to speed-up the bootstrapping process. The new node is not allowed to request any service until it is admitted. Each transaction, both from the new and old nodes are signed. We count on the reputation of the introducing nodes to expect that they will act fairly to the newcomers. However, they can loose reputation for inappropriate introduction, and can also gain for appropriate ones.

When *nodeB* joins through multiple nodes (e.g. *nodeA*, *nodeC*, *nodeE*), *nodeA* determines if *nodeB* is due to be added as a neighbor, by collecting recommended Bootstrap factors (*Bf*) from the other nodes. *NodeA* further weighs each recommendation based on its trust on the nodes that sent them, as illustrated in Eq. (3). *BfCB* is the *Bf* reported by *nodeC* concerning *nodeB*, while G*SAC* is the reputation of *nodeC* from the perspective of *nodeA*. Same pattern repeats for every neighbor that would submit Bf recommendation. Notice that *nodeA* still adds its own bootstrap factor, but with an optimal self-reputation of 1:

$$\mathrm{Bf}_{\mathrm{A_B}} = (\textstyle\sum_C \mathrm{Bf}_{\mathrm{CB}}.\mathrm{GS}_{\mathrm{AC}}) + \mathrm{Bf}_{\mathrm{AB}}. \tag{3}$$

No further normalization is relevant at this stage because the newcomer is not expected to service too many nodes before it joins, and not all neighbors are therefore expected to respond to a request for *Bf* recommendation. Notice also that the interest is on the cumulative value, not the average. We pass the accumulated *Bf* through a unit step function, *Θ(Bf)* which returns either 0 or 1 depending on the value it receives, according to Eq. (4), where n_{min} is the minimum *Bf* required from a newcomer:

$$\Theta(\mathrm{Bf}) = \begin{cases} 0 \text{ if } < \mathrm{n_{min}} \\ 1 \text{ otherwise} \end{cases}. \tag{4}$$

The self-signed certificates mentioned previously, does not stop generation of multiple fake identities, but it helps to ensure that malicious nodes do not impersonate existing trustworthy ones. Recall that new nodes usually do some work in order to gain entry into the network, which is to service some existing node(s) up to a minimum limit. This would mean spending some resources and a bit of time by the newcomers. Since resources are not always limitless, the number of fake identities that a Sybil attacker can introduce into the network is limited. The attackers are further frustrated in the network through the reputation and 'interaction rate' checks which are subsequently discussed.

It is important to minimize the number of fake identities at entry stage because the lesser they are in number, the lesser their impact. The difference between the work done by newcomers in FBit and that which they do by solving puzzles, is that genuine FBit nodes which are already in the network do not have to spend extra resources to verify the work done by newcomers. They simply perform normal checks which they usually do for all massages, even those from known trustworthy nodes. So, there is less burden (due to bootstrapping) on the trustors and the network in general. Whitewashing also appears less appealing and thus nodes are discouraged from becoming malicious after joining the network.

Similarly, the new approach does not suffer the kind of fundamental problem [55] experienced in puzzle based methods, which has to do with disparity in resources or computation abilities between legitimate users and would-be attackers. With the new method, even large disparity in computational abilities between legitimate nodes and the new comers (or potential attackers) is not of visible significance. Moreover, the proposed method favors productivity, because newcomers actually render valuable help, and in return gain initial reputation. As explained in subsection 5.4, this method can be adapted to similar other file sharing P2P platforms. It is distributed, and does not rely on any seed node or super peer.

5.2 Familiarity

Many P2P file sharing systems (e.g BitTorrent) operate in a "give and take" manner. This is not strange since the network is usually self-sustaining. The idea behind familiarity (or interaction rate) is to enable nodes to ascertain how much each

neighbor has contributed to them particularly, and to the network in general. We assume that more familiar nodes interact more often, and if genuine, contribute more to each other. This idea is similar to the tit-for-tat method of BitTorrent where nodes prioritize neighbors that have given more to them.

However, unlike BitTorrent where nodes act mainly based on local tit-for-tat views, the new method allows nodes to also ascertain the global impact of each node on every other node in the network (or neighborhood) at a given time, in-terms of resource contribution, on a scale of 0 to 1. This ensures that a more liberal and reputable node, gains some advantage, not just for its reputation but also for its liberality in terms of rendering services to other nodes. We shall focus on familiarity in this subsection, and then capture reputation in the next.

Equation (5) captures the activity rate of peers in a swarm (which is a kind of P2P network) at intervals ($t = 20$ s). If node i downloads from node j, or uploads to it, i records an interaction (download or upload), and vice versa. We also take note of unanswered requests, because they could be signs of maliciousness, such as lying piece attack. The time when interactions occur are also recorded. Given that node i has x successful interactions (or transactions) with node j within time t, it calculates the interaction rate of node j; $IR_{ij}(t)$ using Eq. (5):

$$\mathrm{IR_{ij}(t)} = \mathrm{X_{ij}/n_{max}} \tag{5}$$

where n_{max} is derived by dividing the node's bandwidth by the size of each block and multiplying the result by time (t). Literally, n_{max} is the optimal rate at which each peer is expected to function in the swarm within t interval, and it is recalculated every t seconds. An alternative way to determine n_{max} in an environment where knowing the bandwidth is difficult, would be to collect votes from participating nodes based on their interactions during their first few seconds in the network. Nodes pay more attention to neighbors who service them in return. If extended, it means that nodes give priority to neighbors who give back to the network, in order to ensure sustainability. Successful transaction means non malicious uploads and downloads, which are expected to be considerably mutual.

As familiarity grows, IR tends to 1. n_{max} is chosen in a way that keeps IR below 1 within the time (t) range. However, in cases of extreme familiarity, depending on the choice of n_{max}, IR can be greater than 1. When this happens (which can be rare in practice), the excess is not considered, so IR simply equals 1.

As already noted, most earlier approaches use parameters such as the difference between number of a peer's *'downloads from'* and *'uploads to'* another peer as a way of determining cooperation. For instance, if node i has uploaded z chunks to node j and downloaded y chunks from it, then y minus z will be the bases for determining the fairness or otherwise of j, from the perspective of i. In some cases, ratio is used instead of the difference, mostly in models that are based on probability expectation value [23]. y and z can also represent the number of favorable and unfavorable interactions respectively. Unanswered requests are often counted among the unfavorable transactions.

The problem with such method is that the cumulative interaction rate does not reflect adequately on the end results. As an example, consider a case where $y = 500$, and $z = 499$, the difference will be 1, just the same as when $y = 2$ and $z = 1$, ignoring the interaction frequencies, which can be a vital factor by itself. If we take the ratio of good interactions over total, then the case with fewer interaction will be at clear advantage, which can be somewhat unjust. Wang P. et al. [56], identified similar problem, but their approach soldered the concept rigidly into trust computation, such that it might be a problem when emphasis is preferred either on just trust or interaction rate.

In a nutshell, we use this concept of familiarity to capture the activity rate of a peer, not just locally, but generally in the network, on the scale of 0 to 1. Trust score alone can hardly reveal this relationship. We have applied it more specifically in subsection 5.5 to tackle bandwidth attack.

5.3 Reputation score computation

Applying the concept of probability expectation value [23], and based on recorded interaction experience, non-malicious downloads are associated with α, while the bad ones plus uploads (as well as no replies) are associated with β. Expected behavior based on the previous reputation (DT_{ij}) of node j, according to the direct experience of node i with respect to time (t), is given in Eq. (6):

$$\mathrm{DT_{ij_{(t)}}} = \frac{\alpha_{\mathrm{ij_{(t)}}} + \Theta\left(\mathrm{Bf_{i_j}}\right)}{\alpha_{\mathrm{ij_{(t)}}} + \beta_{\mathrm{ij_{(t)}}} + \Theta\left(\mathrm{Bf_{i_j}}\right)}. \tag{6}$$

A download is considered non malicious if its piece is hashed without error. Otherwise, if there is a mismatch in the hash code, it is considered malicious. Bf serves as a normalizing factor when computing expected reputation, since every node in the neighborhood of i is expected to have been bootstrapped, which implies that it has $Bf > 0$. The number of 'uploads' counts for β because we are interested in both credibility and contribution rate. For example, the reputation of a free-rider can drop as a result of its act of not giving. While those that barely give, would be caught up in the familiarity check.

As the number of interactions grow, the behavior of nodes may change, making it important for older interactions to have less weight or be completely forgotten. After more than 1 interactions (i.e. if $\exists\ DTij(t-1)$), decay or aging factor is introduced. The aging factor used here is related to that of [8] and it basically considers the similarity between previous records and the current outcome, the wider the difference, the less consideration such history is given. If we denote the aging factor by $\rho^{(t)}$, then we update our local experience with respect to the just concluded transaction at time (t), as given in Eq. (7):

$$DT_{ij_{(t)}} = \rho^{(t)} DT_{ij_{(t-1)}} + \left(1-\rho^{(t)}\right) DT_{ij_{(t)}}, \tag{7}$$

this equation also applies to IR.

To get a general view of the reputation and familiarity of any neighbor, nodes usually ask others for recommendations concerning that neighbor. Replies to such recommendation requests usually come in pairs; DT and IR. Considering our testbed (BitTorrent), nodes can also update the tracker with recommendation information when they make contacts, although such tracker update is not a requirement for the proposed method. When a node asks for recommendation from its neighbors, it ranks each recommendation according to the reputation of the recommending node, using the Ordered Weighted Average (OWA) [57]. The choice of OWA is due to its weight tuning advantage, it allows us to easily associate each indirect score with the reputation of the node that sent them. If node i asks its n sequence of neighbors (j) about another peer l, i discounts collected recommendations (DT_{jl}), and interaction rates (IR_{jl}) as indicated in Eqs. (8) and (9):

$$IT_{il_{(t)}} = \frac{\sum_{j=1}^{n} DT_{ij_{(t)}} DT_{jl_{(t)}}}{\sum_{j=1}^{n} DT_{ij_{(t)}}}, \tag{8}$$

$$CIR_{il_{(t)}} = \frac{\sum_{j=1}^{n} IR_{ij_{(t)}} IR_{jl_{(t)}}}{\sum_{j=1}^{n} IR_{ij_{(t)}}}, \tag{9}$$

where $IT_{il_{(t)}}$ and $CIR_{il_{(t)}}$ are the reputation and IR of l respectively, according to i, based on the information it got from other peers. And assuming that $DT_{jl_{(t)}}$ scores are arranged in descending order. Similarity check, elaborated in section 6, is performed on the collected recommendations before computation. This is to sieve out recommendations that may have been submitted by collusion attackers.

The trustor's local reputation about the trustee $\left(DT_{il_{(t)}}\right)$ can be added using Eqs. (8) and (9), with peak weight of 1. Alternatively, recommended scores (IT, CIR) and direct scores (DT, IR) can be distinct, and then merged with Eqs. (10) and (11), adjusting the weight ($0 \le \sigma_d \le 1$) to suit peculiar needs. We adopted the later (using Eqs. (10) and (11) for the results shown in this work, with a weight of 0.6 assigned to σ_d. In our experience, this value appears optimal because it gives reputation an upper hand, without undermining familiarity factor:

$$TR = \sigma_d.DT + (1-\sigma_d).IT, \tag{10}$$

$$TIR = \sigma_d.IR + (1-\sigma_d).CIR, \tag{11}$$

where TR stands for Total Reputation, and TIR is the Total Interaction Rate. They can further be merged with Eq. (12). We observed on the course of this work that it makes lots of difference when IR is considered, compared to when it is not. ($0 \le \sigma_t \le 1$) is a weighting factor, in Eq. (12):

$$GS = \sigma_t.TR + (1-\sigma_t).TIR, \tag{12}$$

σ_t has same value as σ_d mentioned previously.

5.4 Modified BitTorrent unchoke algorithm for leechers

For clarity, we have divided the algorithm into two parts. The first (algorithm 1) briefly discussed in this subsection, captures the steps that FBit leechers take when unchoking other leechers. While the second part (algorithm 2) which is presented in the next subsection, depicts the steps that seeders take when unchoking leechers. Algorithm 1 applies earlier discussed concepts of bootstrapping, familiarity and reputation to mitigate Sybil attacks, fake-block attacks, free-ridding, and similar others. While algorithm 2 focuses specifically on stopping bandwidth attacks. The two algorithms work smoothly together.

Some acronyms used in the algorithms which were not earlier defined are:

- *MaxUnchoke*; used to indicate the maximum number of neighbors a server node can service simultaneously at a given time.
- *NumberUnchoked*; used to show the number of neighbors that is currently being served by a given node.
- *InterestedPeerList*; keeps a list of neighbors that are interested in a given block of the file.
- *TrustedPeerList*; used to keep a list of nodes (among those interested in the current block) whose reputation score is above *'threshold'*. IR does not influence this score; *'GS'* only counts for peers that have not been found malicious at this preliminary stage.
- *threshold*; minimum score (0.5) a peer must have before it can be added to the list (*'InterestedPeerList'*) of those that will be considered for a given transaction.

Algorithm 1 Leecher Unchoke Method

```
1: MaxUnchoke = c
2: NumberUnchoked = 0
3: InterestedPeersList ← P
4: TrustedPeerList ← {}
5: while NumberUnchokded ≤ MaxUnchoke do
6:     for all peers(P) in InterestedPeerList do
7:         Calculate IR and DT scores()
8:         if IR Is High then
9:             TR = DT
10:            TIR = IR
11:        else
12:            Get recommendations and check similarity
13:            Apply equations (8) to (11)
14:            Calculate GS() according to (12)
15:        end if
16:        if TR ≥ threshold then
17:            TrustedPeerList ← p
18:        end if
19:        Sort TrustedPeerList by GS()
20:        Unchoke Top Scores First()
21:        NumberUnchoked++
22:    end for
23:    Remove P from InterestedPeersList
24: end while
```

In a nutshell, here is the function of each line in the first algorithm. Lines 1 and 2 initiates *MaxUnchoke* and *NumberUnchoked* variables respectively, while lines 3 to 4 start the lists of *InterestedPeerList* and *TrustedPeerList*. According to line 5, if a node is currently able to service any additional neighbor, it selects a trustee from the *InterestedPeerList* (line 6) and calculates the *IR* and *DT* of the trustee (line 7). If the trustee is very familiar to the trustor (line 8) then scores are based on direct experience only (lines 9 and 10), otherwise recommendations are collected (lines 12) and used for calculating scores (lines 13). When recommendations are collected, a similarity check (explained in the last paragraph of section 6) is also done. *GS* is afterward calculated in line 14. However, if a trustee has a TR that is below threshold, it does not qualify to be considered for any service (lines 16 to 18). Those who are qualified to receive services are serviced according to their *GS* scores beginning with the highest score (lines 19 and 20). When a transaction process is initialized, the trustee is removed from the *InterestedPeerList* (line 23).

When nodes are able to validate each other and complete transactions, they mutually update reputations and interaction rates, reflecting their experiences. In algorithm 1, leechers use such updates in determining requests to respond to, and the priority that each requester deserves. Equations (6) and (7) are applied to update direct reputation and familiarity (rate) information. If a node has communicated regularly with consistent reputation for a considerable amount of time (even recently), then it makes sense to apply local information in making decision for a subsequent transaction, in order to save resources that would otherwise be used to gather and compute recommendations.

Trustors contact neighbors for recommendations when they do not have enough information to assess the trustee, or when the direct familiarity between the two nodes is not high enough. Line 13 of algorithm 1 triggers Eqs. (8) to (11) which are responsible for computing indirect scores and combining them with direct experience of the trustor. The general score (GS) is further calculated using Eq. (12) which basically applies desired weights to merge reputation and familiarity scores. Based on the GS, all trustees that are not considered malicious are collected in a list and priority is given to each according to its contribution and reputation in the network.

Although peers may be given different names in different P2P platforms, the concept of seeder (service giver) and leecher (service receiver) is quite common. In most cases, especially in distributed P2P, network peers serve as both service providers and consumers. The algorithms presented in this work has not been built to rigidly fit into BitTorrent topology alone. Focus has been more on the nodes, so that any similar topology would require minimal tuning to adapt it.

5.4.1 FBit in DHT based systems

DHT (including MLDHT) makes it possible for the network to function without a tracker. Every node acts as a mini tracker; they collectively perform the task of discovering other peers in the network. In MLDHT, nodes randomly choose 160-bit unique ID which also indicates their distance (in a way). At first, nodes need to query the k-bucket to get nodes that are closest to the infohash. The client node further tries to connect to those nodes in order to initiate queries. Given that the connected node is aware of the peers that are associated with the infohash, it returns a list of such peers and file download continues similar to the way it happens in the traditional protocol [58].

At this point, just before the download processes, the reputation bootstrapping can happen for newcomers. Key exchange messages can be embedded in any of the controls mentioned earlier (eg. 'FIND_NODE'), while newcomers may be served after successfully responding to some requests such as request for peers that are associated with the infohash (which the newcomer should have at this stage). In the current implementation of the proposed system, the tracker does not perform any special task, every node is regarded as equal. This is a way of making it non-centralized and enabling its adaptability to other platforms.

FBit is focused more on content downloads (retrieval/storage) and less on routing, but it can be featured in both processes. Instead of considering only the distance in choosing the k-bucket nodes, *GS* score in FBit can be engaged in the process. A weighted average of the *GS* and "distance factor" can be adopted, similar to the method used in [43]. With the advantages being reputation bootstrapping, computation of indirect reputation and mitigation of bandwidth attack.

Similarly, in Gnutella, servants perform the task of both servers and clients [59]. To join, Gnutella newcomers connect

to any known host, and through that means get in-touch with other nodes. Such known host(s) can serve as the initial trustor(s) in the bootstrapping stage of the proposed system. Similarly, the "broadcast" and "back-propagation" of information in Gnutella can serve as means of relating the reputation (and other information) of peers to neighbors within the network. The new method can also be adapted easily to fit into other offspring of P2P such as mobile edge-clouds since every node is capable of making independent (and informed) decision.

5.5 Modified BitTorrent unchoke algorithm for seeders

As mentioned earlier, malicious peers take advantage of the opening discussed under bandwidth attack to abuse seeders and download mainly from them, thereby blocking their unchoke slots (i.e. making them unable to upload to legitimate peers). Such behavior also enable the malicious nodes to avoid downloading from other leechers who could measure their contributions based on experience. To check this, FBit in algorithm 2 warrants that a leecher (l) who requests download from a seeder (s), has to submit its top *IR* information alongside its request. s uses such information to determine how much l has uploaded and downloaded from its fellow leechers.

The seeder then gives priority to peers that are more selfless in their service to other leechers, in order to motivate them and encourage others not to be selfish. It also mitigates a kind of denial of service attack that would occur if the unchoke slots of seeders are congested by attackers. This algorithm further holds each peer responsible for not only its reputation, but also the impact it is able to make in the network. Recall that seeders are basically service givers who have no need for receiving. P2P networks need them, and they need to be encouraged and optimally protected.

Algorithm 2 Seeder Unchoke Method

```
1:  MaxUnchoke ← {predefined}
2:  InterestedPeersList ← {}
3:  NumberUnchoked ← 0
4:  while NumberUnchokded ≤ MaxUnchoke do
5:      CollectVotesIncludingRecentlyCached()
6:      DoIntermitentScoreValidation()
7:      CountVoteForEachP()
8:      UnchokeHighestVotedP()
9:      if PeersHaveSameVote() then
10:         UnchokePWithHigherIRScore()
11:     end if
12:     NumberUnchoked++
13:     RemovePFromInterestedPeersList()
14:     if VotedP Is Not In InterestedPeersList then
15:         CacheVote()
16:     end if
17: end while
```

As an illustration, nodes l, i, and j in Fig. 1 represent individual peers in a swarm who request services from a seeder node. The message accompanying their requests indicate other peers (leechers) that they have downloaded the most from, and the ones they have uploaded the most to. A malicious node who chooses to download only from seeders in order to dodge assessment by fellow leechers, will have little or no "upload" information to give and thus will not be qualified to make requests. In this illustration, node i will be served first because it has the highest vote, before node l and then node j.

Seeders confirm scores by validating submitted scores for consistency. For instance, it could be seen that nodes i and l have both submitted scores about each other at same time interval; it is therefore expected that the download rate (*DR*) score which i submitted concerning l at a given time should match the upload rate (*UP*) score that node l submitted concerning i at about the same time. Line 6 of algorithm 2 executes this check, every t seconds.

If two or more peers have the same number of votes among the submitted top leechers, then the highest IR score will be given priority, according to lines 9 to 11 of the algorithm. If their *IR* scores are still same, the first request will be served first. By giving priority to nodes that show more cooperation with fellow leechers, they are rewarded for being cooperative and others are encouraged to do likewise. Low score nodes which are genuine will have no problem downloading from other leechers in order to step up their scores. At the initial stage however, services by seeders are on first-come, first-served basis.

Seeders also cache some of the information temporarily, so that it could be used in the near future if need be. For example, if any other node besides i, j and l were voted, its score will be saved for when such node would request a service, within some time limit (lines 14 to 16 of algorithm 2).

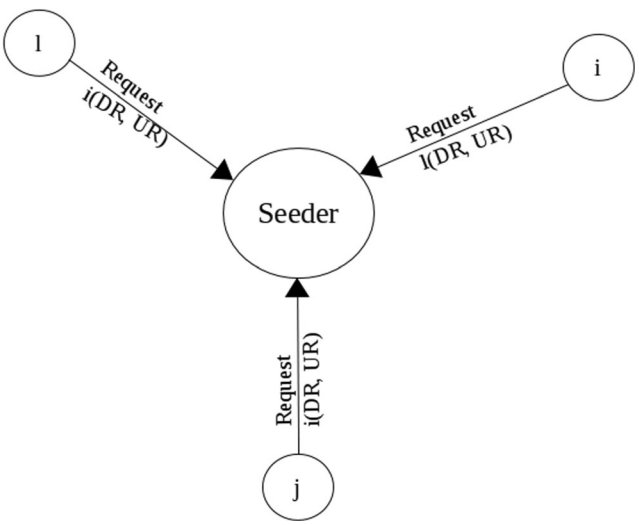


Fig. 1 When leechers send requests to a seeder

Here is a summary of the functions of the remaining lines in algorithm 2. Lines 1 to 3 initiates the variables the same way it is done in the first algorithm, and line 4 also has similar function. Line 5 enables a server node to collect votes from its client nodes indicating the *IR* information of other nodes they have interacted with. Line 7 counts valid votes for each client node in the *InterestedPeerList* and then services are rendered based on active cooperation in the network (line 8). Once a transaction process is initiated, line 12 updates the number of nodes that are currently being served and line 13 removes the node from *InterestedPeerList* to indicate that it is being served. After each transaction round, the *MaxUnchoke* is decreased.

6 Experiments and results

The PeerSim simulator was used to implement a testbed for the proposed method. It is a java based P2P simulator which already includes a BitTorrent protocol implementation [14, 60]. This allowed us to focus only on the adaptation of the protocol to our approach, while retaining the original version for comparison.

The simulation runs on two virtual machines each with 16 CPUs at 2500 MHz and 64GB RAM. Each machine runs OpenJDK Runtime Environment 1.8, and Python 3.4.3 which is used for scripting. PeerSim has a configuration file that allows parameters to be adjusted as desired. Some of such parameters include;

- *network.size;* used to specify the network size. This was set to 100.
- *network.node.direct_weight*; this is a new addition, used to specify the weight of direct scores (that is, *IR* and *DT*). It was set to 60%.
- *protocol.urt* was set to "UniformRandomTransport", which is the transport protocol used by the simulator.
- *protocol.simulation.file_size*; this is used to specify the size of file to be shared in the network. The process is completely successful if every peer downloads a complete file before the process ends. It was set to 20mb in the simulation.
- *protocol.simulation.duplicated_requests*; this was set to 1, meaning that a node can only send one request for a particular block at a time. It could be set to any other value if desired.
- init.net.*seeder_number*; this was set to 1, so that each simulation begins with one seeder. There are other variables that are initialized at the start of the simulation such as the number of attackers. When a type of attack is not present, its value is set to 0 (e.g. init.net. *nCollusionAttacker 0*). There are also some controls that allow us to observe and get feedback from the simulation (e.g. control.observer.step simulation.logtime). During "logtime", we update the files that have been created to store the information needed to measure the performance of the network.
- *protocol.simulation.max_swarm_size* was set to 200 to show that the network can only grow to a maximum of 200 nodes (there is no specific reason for choosing 200, except that we needed a number that is more than the network size which is 100). This can change to accommodate any network size of choice.
- *random.seed*; every simulation needs to have a seed value. Using the scripts, this and other necessary variables are automatically generated at the beginning of each simulation. In the case of the random seed, the script generates it randomly, while other values (such as percentages of attacker) are picked accordingly from the series of provided values.

A 95% confidence level was maintained for the recorded simulation results, this is in order to statistically validate their consistency. Simulations were ran with network size of 100 nodes, and varied number of malicious nodes, to determine the effect in each case. Different attacks were simulated including sibyl, fake-block and collusion attacks, with the goal of determining how efficiently such attacks are mitigated by FBit compared to other methods. The original BitTorrent method [60] and another Trust Management System (TMS) [3] were adopted for comparison. For TMS, we implemented it using the information provided in [3]. TMS was chosen for comparison because it addressed similar attacks (such as fake-block attack), and was also tested on BitTorrent platform.

The simulation began with a single seeder in each case, and runs until a 95 percentage confidence level is attained, after which an average result is gathered. Each simulation can run for a maximum of 30 times. The file size is 20mb for all experiments. This work did not address the implications related to churn, nodes stay in the network throughout the simulation. The presented results were obtained with network size 100 and varied percentages (0, 10, 20, 30, 40, and 50, 60, 70, 80) of attackers.

Figure 2 presents a result of how the network responded to Sybil attackers, who also distribute fake-blocks when they are able to gain access. The result compares FBit to the original tit-for-tat based BitTorrent and TMS. As shown, non-malicious nodes suffer less attack with FBit compared to other methods; they are able to download at a significantly faster rate. The proposed bootstrapping method helps to minimize the introduction of fake identities (IDs) into the network. Introducing a fake ID is made costly and thus a Sybil attacker can only introduce a limited number, based on its capacity. FBit further fishes out such limited number of fake identities through reputation and familiarity checks.

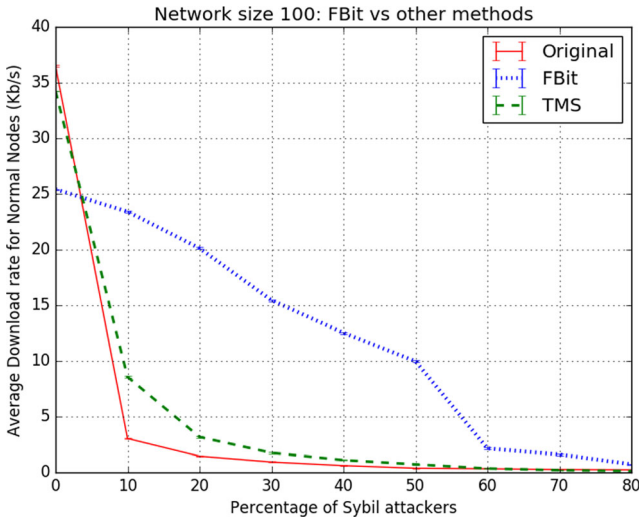


Fig. 2 FBit vs other methods; showing the difference in download rate for non-malicious nodes. FBit allows them to download at higher rate amidst Sybil attackers

When the number of attackers rose beyond 50% of the network size, the graph (Fig. 2) indicates that their effect on the FBit network became stronger, and mitigating them led to a sharp decrease in the download rate of legitimate nodes. This is expected because, with such high percentage, there will be more failed and repeated transactions. Transactions fail when nodes suspect that their neighbor is likely an attacker, and decline its offers. When this repeats, more time will be needed to find the right nodes, and thus the overall transaction time will increase.

Reputable nodes which are already in the network do not share in the cost of introducing or validating newcomers, except for slight 'transaction time trade-off' that may arise as a result of routing some chunks or queries through the newcomer. Such tradeoff is however compensated by the gain in reputation that the trustor stands to get after successfully introducing a new node. The delay caused by forwarding some packets (or queries) through the newcomers appears mild. If legitimate existing nodes had to spend computational resources for challenge verification, as would be the case in puzzle based methods, the impact would be more significant (at-least from logical point of view).

TMS could not cope with Sybil attackers because of the earlier mentioned flaw in the bootstrapping method. If fake IDs can be introduced almost at no cost, then an attacker can easily introduce them in numerous number and hijack the network at early stage. In tit-for-tat, new comers are admitted without checks via optimistic unchoke, giving room for attackers who sap the available resources and frustrates the network with fake blocks.

FBit was also exposed to collusion attacks, where fake-block attackers cooperate to favor themselves and downgrade the reputation of others that are not in their clique. Figure 3 captures the performance of FBit in comparison with tit-for-tat based BitTorrent and TMS. A careful look at the result reveals that FBit can cope with collusion attacks. Although collusion attackers appear to have higher impact compared to a non-collusion scenario such as Fig. 2, FBit nodes are still able to complete download at considerably fair rate. The noticed decline in download rate (with higher percentage of attackers) is expected, because finding reliable nodes become more difficult and more time consuming. The amount of genuine pieces available in the network also declines as the number of genuine nodes decrease. The TMS method felt more collusion impact, while the original method was completely overwhelmed.

To guard against collusion attack, TMS uses recommendations from top 10 nodes to compute the scores of the other peers. From the result shown, this does not appear effective because malicious nodes also stand a chance of joining the top nodes, especially with exaggerated scores from other colluding nodes. Tit-for-tat also does not effectively combat collusion attack because nodes are overwhelmed by malicious traffic from the attackers. For example they can be trapped in a cycle of downloading, verifying and discarding fake blocks; making it impossible for them to successfully complete transactions. Moreover, in TMS and original methods, attackers are able to steal resources from seeders through bandwidth attack.

Some models have applied neighbor similarity to detect attackers in P2P. For example, in [61], if recommendations collected from various neighbors concerning a peer, lack similarity, then such peer is regarded as an attacker. Their approach was designed for e-commerce, and it requires coordination from some form of central agents such as group leaders, which can be seen as a limitation when considering more distributed platforms. However, we tapped from their idea of similarity, and applied it (with modifications) to tackle collusion attack.

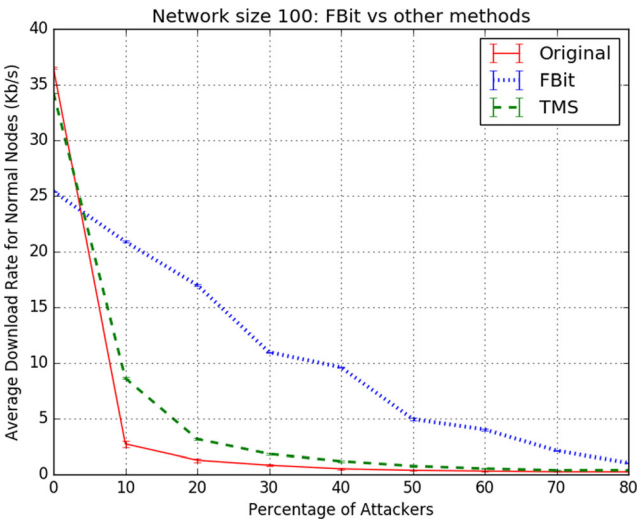


Fig. 3 Collusion attack

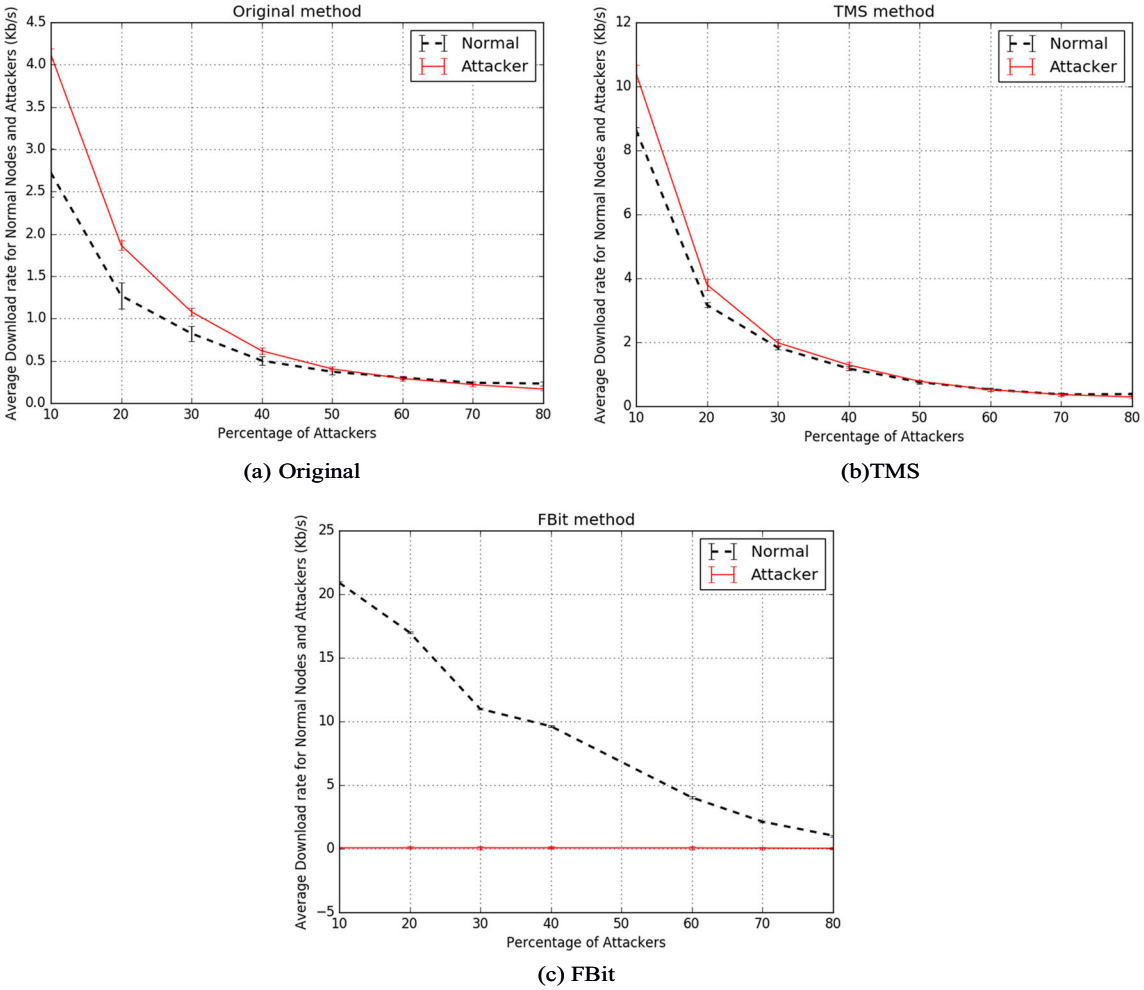


Fig. 4 The download rate of genuine nodes versus attackers in the original (**a**), TMS (**b**) and FBit (**c**) methods. A Resilient system is expected to frustrate attackers and allow non malicious nodes to download faster. In the original and TMS methods, attackers are able to download, especially from seeders. TMS did better than the original method, but was still tricked by the attackers. FBit successfully stopped the attackers from stealing resources (from seeders), showing resilience against bandwidth attack

When a peer is sending a request for recommendation, it includes a randomly selected trusted neighbor(s) from its neighbor list (which must be different from the neighbor being inquired about, and the one from whom recommendation is required). So it requests recommendation for the actual unfamiliar peer, and at least one familiar fellow that would be used to check similarity.

If the recommendation giver is in a collusion clique with the unfamiliar peer, then it will give it high score, and downgrade the reputable one. A malicious recommendation giver can also decide to simply exaggerate both scores, but it will still be noticed in the similarity check. When recommendations from various neighbors are gathered, priority is given to the ones that are most similar, based on the known peers that have been inquired about. Line 12 of algorithm 1 triggers the similarity check.

To measure how successful an attack is, we captured how the malicious nodes fared compared to non-malicious ones. Ideally, non-malicious nodes are expected to beat the attackers and maintain a clearly higher download rate. Figure 4 illustrates this, with non-malicious nodes downloading clearly at higher rate in FBit, while the collusion attackers were dominating in the other methods. The gap is more in the original method and less in TMS, indicating that TMS is more resilient to collusion attack than the original method.

7 Conclusion

We have devised a bootstrapping approach for BitTorrent and similar P2P protocols. It is distributed and appears efficient with the simulation test cases. In contrast to the popular method of assigning default reputation scores to newcomers, our method provides them with a distributed avenue to work for their initial reputation, before they are able to fully join the

network and request services. No central entity or pre-existing relationship is required.

Similarly, the proposed algorithm adequately shields seeders from the effects of bandwidth attacks. This is possible through a modified familiarity approach that makes seeders aware of peers' leecher-to-leecher relationships. Seeders are important asset in P2P and need to be protected adequately, but as we highlighted, they receive minimal attention before now and therefore they face exploitation. The proposed model promises improved network stability, security, fairness and efficiency.

We are continuing research on distributed P2P along several directions. As a follow up to this work we are exploring the concept of Root Cause Assessment (RCA) [62], to see how they can empower nodes to deal with irregular behaviors, such as cases where nodes alternate between good and bad behaviors irregularly. There is also an ongoing plan to fit FBit into an edge cloud platform such as [63, 64].

Acknowledgements This work is partially funded by project "NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics/NORTE-01-0145-FEDER-000016" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

References

1. Khan, A.M., Freitag, F. Rodrigues, L.: Current trends and future directions in community edge clouds. In: 4th IEEE International Conference on Cloud Networking (CloudNet), pp. 239–241. IEEE, Niagara Falls (2015)
2. Baqer K, Anderson R (2015) Do you believe in tinker bell? The social externalities of trust. In: Cambridge international workshop on security protocols, pp. 224–236. Springer
3. Sarjaz BS, Abbaspour M (2013) Securing BitTorrent using a new reputation-based trust management system. Peer-to-Peer Networking and Applications 6:86–100
4. Konrath, M. A. Barcellos, M. P. Mansilha, R. B. : Attacking a swarm with a band of liars: evaluating the impact of attacks on bittorrent. In: 7th IEEE international conference on peer-to-peer computing, pp. 37–44. IEEE (2007)
5. Naghizadeh A, Razeghi B, Radmanesh I, Hatamian M, Atani RE, Norudi ZN (2015) Counter attack to free-riders: filling a security hole in BitTorrent protocol. In: 12th IEEE international conference on networking, sensing and control, pp. 128–133. IEEE
6. Dhungel P, Wu D, Ross KW (2009) Measurement and mitigation of BitTorrent leecher attacks. Comput Commun 32:1852–1861
7. Wong KY, Yeung KH, Choi YM (2009) Solutions to swamp poisoning attacks in BitTorrent networks. In: 1st international MultiConference of engineers and computer scientists, pp. 360–363. IMECS
8. Aringhieri R, Damiani E, Vimercati D, De Capitani S, Paraboschi S, Samarati P (2006) Fuzzy techniques for trust and reputation management in anonymous peer-to-peer systems. J Am Soc Inf Sci Technol 57:528–537
9. Qureshi, A. Rifa-Pous, H. Megıas, D.:Electronic Payment and Encouraged Cooperation in a Secure and Privacy-Preserving P2P Content Distribution System. In: The 7th International Conferences on Advances in Multimedia, pp. 8–14. MMEDIA(2015)
10. Kamvar SD, Schlosser MT, Garcia-Molina H (2003) The eigentrust algorithm for reputation management in P2P networks. In: Proceedings of the 12th international conference on world wide web, 640–651. ACM press
11. Cornelli F, Damiani E, di Vimercati S, Paraboschi S, Samarati P (2002) Choosing rep- utable servents in a P2P network. In: Proceedings of the 11th international conference on world wide web, pp. 376–386. ACM press
12. Dhungel P, Hei X, Wu D, Ross KW (2008) The seed attack: can bittorrent be nipped in the bud?. Technical report, Department of Computer and Information Science. In: Polytechnic institute of NYU
13. Dhungel P, Hei X, Wu D, Ross KW (2011) A measurement study of attacks on bittorrent seeds. In: 2011 IEEE international conference on communications (ICC), pp. 1–5. IEEE
14. Montresor A, Jelasity M (2009) PeerSim: A scalable P2P simulator. In: 9th IEEE international conference on peer-to-peer computing, pp. 99–100. IEEE
15. Nwebonyi FN, Ani UP (2015) DanielBYOD network: enhancing security through trust– aided access control mechanisms. International Journal of Cyber-Security and Digital Forensics 4:272–290
16. Gambetta D (2000) Can we trust trust?. Trust: making and breaking cooperative relations. In: Gambetta, Diego (ed.) trust: making and breaking cooperative relations, electronic edition, Department of Sociology, University of Oxford, pp. 213–237. University of Oxford
17. Jøsang A, Ismail R, Boyd C (2007) A survey of trust and reputation systems for online service provision. Decis Support Syst 43:618–644
18. England P, Shi Q, Askwith B, Bouhafs F (2012) A survey of trust management in mobile ad-hoc networks. In: Proceedings of the 13th annual post graduate symposium on the convergence of telecommunications, networking, and broadcasting. PGNET
19. Lilien L, Al-Alawneh A, Ben Othmane L (2010) The pervasive trust foundation for security in next generation networks. In: Proceedings of the 2010 workshop on new security paradigms, pp. 129–142. ACM
20. Resnick P, Zeckhauser R, Swanson J, Lockwood K (2006) The value of reputation on eBay: a controlled experiment.: experimental economics, pp 79–101. Springer
21. Gregg DG (2009) Outline reputation scores: how well are they understood?: journal of computer information systems, pp 90–97. Taylor & Francis
22. Venkanna U, Agarwal JK, Velusamy RL (2015) A Cooperative Routing for MANET Based on Distributed Trust and Energy Management. In: A cooperative routing for MANET based on distributed trust and energy management.: wireless personal communications, pp. 961–979. Springer
23. Josang A, Ismail R (2002) The beta reputation system. In: Proceedings of the 15th bled electronic commerce conference, pp. 2502–2511. Bled
24. Skopik F, Schall D, Dustdar S (2009) Start trusting strangers? Bootstrapping and prediction of trust. In: International conference on web information systems engineering, pp. 275–289. Springer Berlin Heidelberg
25. Yahyaoui H, Zhioua S (2011) Bootstrapping trust of web services through behavior observation. In: International conference on web engineering, pp. 652–659 springer Berlin Heidelberg
26. Sherchan W, Loke SW, Krishnaswamy S (2006) A fuzzy model for reasoning about reputa- tion in web services. In: Proceedings of the 2006 ACM symposium on applied computing, pp 1886–1892. ACM