BAKUP: Automated, Flexible, and Capital-Efficient Insurance Protocol for Decentralized Finance

Srisht Fateh Singh
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
srishtfateh.singh@mail.utoronto.ca

Panagiotis Michalopoulos
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
p.michalopoulos@mail.utoronto.ca

Andreas Veneris
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
veneris@eecg.toronto.edu

***Abstract*—This paper introduces BAKUP, a smart contract design that insures decentralized finance users against the vulnerability risks in third-party platforms. Apart from providing an automated claim payout, the modular structure of BAKUP brings harmonization among three conflicting features: resilience against vulnerabilities, flexibility of the underwritten policies, and capital efficiency. An immutable core module performs basic accounting while ensuring robustness against external vulnerabilities; a customizable oracle module enables the underwriting of novel policies, and a peripheral (optional) yield module allows users to independently manage additional yield without interfering with the risk management of fellow participants. User payoff is implemented using binary conditional ERC20 tokens tradable on automated market maker (AMM)-based exchanges.**

***Index Terms*—smart contract, decentralized finance, decentralized insurance, risk management, capital efficiency.**

I. INTRODUCTION

Programmable blockchains [1]–[5] have led to an era of decentralized finance, or DeFi, where centralized intermediaries are replaced by software code, also known as *smart contracts*, to provide financial services. This has spawned novel financial innovations such as automated execution, resulting in no market downtime, and permissionless access, increasing user inclusiveness and applications' interoperability compared to its traditional (TradFi) counterpart. This paradigm shift has led to increased utilization of DeFi systems in recent years, with several billion dollars exchanging hands daily [6].

Despite its success, DeFi is not free from risks, thus impeding its rapid adoption [7]–[9]. These risks can generally be classified as *technical* or *economic* [10]. Technical risks arise from exploiting the vulnerabilities of smart contracts, such as programming bugs and/or unintentional functional specification inconsistencies that can lead to irreversible loss of funds held by a DeFi protocol. The second type of risks, stems from economic inefficiencies within a protocol. Technical and economic exploitation, along with external factors, often lead to unexpected behavior in DeFi parameters. This includes fluctuations in token prices and in the liquidity of automated market-making (AMM) pools, as well as extreme borrowing rates in lending protocols[1]. Such deviations can impede the functioning of dependent protocols [11] or result in losses for end users. For example, the liquid staking protocol Lido Finance [12] has a wrapped token for staked ETH (stETH), which is expected to maintain its price stability relative to ETH. However, if this characteristic fails due to the aforementioned reasons, stETH investors may incur unforeseen losses. This emphasizes the imperative need for robust *DeFi risk management tools* to safeguard users from such risks.

One category of solutions to the above problem involves *insurance* where a set of users, known as the underwriters, provides hedging to another, known as policyholders, in return for a fee. However, the very nature of DeFi demands an insurance platform that is (a) automated, ensuring instantaneous claim payout; (b) permissionless in access to achieve interoperability; (c) resilient against both technical and economic vulnerabilities, as it is the last resort for financial adversities; (d) flexible to underwrite policies; and (e) capital-efficient, *i.e.*, any locked capital should be capable of earning yield.

An insurance platform with end-to-end implementation on smart contracts, *i.e.* no off-chain dependencies, achieves the first two properties, *viz.,* automation and permissionless access. However, the latter three merits — resilience, flexibility, and capital-efficiency — *inherently* compromise the objectives of each other. For instance, a high level of resilience in contract design implies high immutability, which in turn hinders the underwriting flexibility derived from adaptability or upgradability. Similarly, the resilience of the smart contract platform may not align well with capital efficiency, as yield generation often involves interactions with and dependencies on third-party protocols that are susceptible to vulnerabilities.

This paper proposes BAKUP, a design for a modular, policy-flexible, and capital-efficient smart contract platform that uses binary conditional tokens to provide insurance. BAKUP allows users and developers to independently balance between resilience, flexibility, and capital efficiency by comprising three distinct modules: a *core module* for basic capital accounting to ensure no defaults, an *oracle module* for underwriting policies, and a *yield module* on the periphery of the above two for capital management. The core module is immutable, while the oracle module can be created unrestrictedly with a custom developer-defined logic, which can range from immutable to highly customizable. Simultaneously, the peripheral yield module operates independently from the main protocol, making it optional for users to engage in yield-generation activities. Interestingly, this optional feature doesn't pose any risk to the capital of users who choose not to participate, allowing users—both underwriters and policyholders—with varying risk preferences to interact simultaneously on the same platform without imposing risks on others.

Here, we first present the design of the core module, followed by a description of the oracle module. Thereafter, we present the yield module's design along with the user incentive mechanism for the module's healthy functioning. Lastly, we evaluate the divergence loss of liquidity providers on AMMs provisioning liquidity for the insurance policies, implemented

[1]https://finance.yahoo.com/news/defi-lenders-spooked-curve-exploit-193953614.html

as a binary conditional ERC20 token. Results show that the above risk can be reduced by over 36% using conservative parameter tuning.

This paper is organized as follows. Section II gives protocol overview, Section III presents the design of the various modules, Section IV evaluates divergence loss, Section V discusses related works, and Section VI concludes this work.

II. Study Motivation and Notational Conventions

A. Protocol Overview

Consider an example where our protocol creates an insurance mechanism for the event of stETH de-pegging in which the price of stETH relative to ETH falls below 0.95. In its simplest version, it creates two tokens: a policy token P and an underwriting token U for a predetermined period, *e.g.,* 1 year. The payoffs of these tokens depend on the occurrence of the underlying event: if the price falls below 0.95 before the deadline, P pays \$1, while U pays \$0. Otherwise, after the deadline passes, P pays \$0, while U pays \$1. For P-holders, it creates a hedging mechanism for the de-pegging event during the specified period, whereas U-holders provide hedging in return for a premium. Although the actual payoffs of P and U in BakUp are slightly different from the above description, the underlying motivation remains the same. The subsequent subsection describes a generalized framework to model financial adversities.

B. Binary Event

We define a *binary event* as a tuple (a, t, s) consisting of:

- Assertion a: A binary event is defined by an immutable assertion a. For the previous example, the assertion is defined as the price of stETH relative to ETH, $p_{\text{stETH/ETH}}$, being less than 0.95 and represented as $p_{\text{stETH/ETH}} < 0.95$.
- Expiration period t: This denotes the time duration since the inception of a binary event during which it remains valid. In the previous example, this was 1 year.
- State s: A binary event can be in either of the two states: *True* or *False*. By default, an event is in the *False* state. If the underlying assertion holds at any instance before the event's expiration, the state shifts to *True* and becomes immutable. Moreover, the state does not change once the event expires.

Therefore, for an adverse effect modeled as a binary event, the occurrence of the $True$ state signifies an insurance claim.

Modelling complex adversities: A binary event only captures a discrete incident. However, in reality, adversities can have varying degrees of damage. To address this, our protocol requires an adversity to be modeled as a collection of $n \in \mathbb{N}$ binary events, where the i^{th} event is represented as (a_i, t, s_i), with all of them having a common expiration.

For instance, a model of continuous incident of stETH de-pegging consists of the following three assertions:

1) $a_1 : p_{\text{stETH/ETH}} < 0.99$.
2) $a_2 : p_{\text{stETH/ETH}} < 0.98$.
3) $a_3 : p_{\text{stETH/ETH}} < 0.95$.

In the above example, if only a_1 holds by the deadline, *i.e.,* $(s_1, s_2, s_3) = (True, False, False)$, it corresponds to a mild de-pegging incident; if both a_1 and a_2 hold, *i.e.,* $(s_1, s_2, s_3) = (True, True, False)$, it represents a moderate de-pegging incident; and lastly, if all the three assertions hold, *i.e.,* $(s_1, s_2, s_3) = (True, True, True)$, it represents an extreme de-pegging incident. Therefore, the final state values together signify the degree of damage.

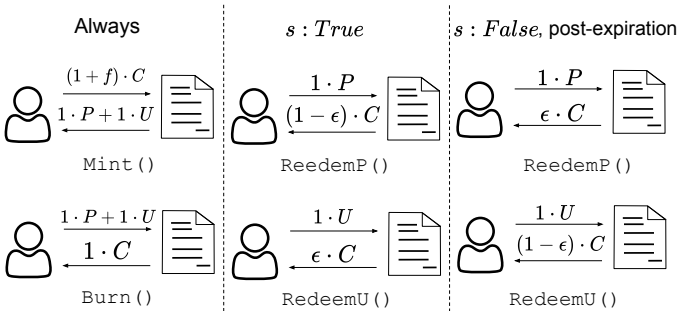


Fig. 1: Interaction between user and core module.

III. BakUp Protocol Design

The protocol design is modular, with key components divided into the core, oracle, and yield modules. Each of these components will be explained next.

A. Core Module

For an adversity modeled as a collection of n binary events, a separate core module contract is created, which in turn creates n pairs of ERC20 tokens P_i, U_i with $i \in \{1 \dots n\}$. Each core contract is characterized by a base currency C (*eg.,* ETH, USDC) in which claims are disbursed. The following module methods can be invoked for each binary event from the inception of the module to perpetuity. Here f represents a constant fee such that $f \in [0, 1)$.

- `Mint(i)`**:** A user depositing $k(1+f)$ units of C, where $k > 0$, receives k units each of P_i and U_i. Here, kf units of C get deposited as fees (incentive) in the `feeTo` address specified by the core module's creator.
- `Burn(i)`**:** A user depositing k units of P_i and U_i receives k units of C.

These methods are depicted in the left part of Figure 1 and they enforce the following two invariants:

Invariant 1. $(1 + f) \cdot C \Rightarrow P_i + U_i$

Invariant 2. $P_i + U_i \Rightarrow C$

Next, we describe methods to redeem the above tokens in exchange for C based on the state of a binary event. At any point in time, $\forall i$ s.t. $s_i = True$, P_i and U_i can be redeemed for the following:

$$P_i : (1 - \epsilon) \cdot C, \quad U_i : \epsilon \cdot C \tag{1}$$

Here ϵ is a constant set by the module creator and is generally close to zero, *i.e.,* $0 < \epsilon \ll 1$. The reason for a non-zero choice of ϵ relates to the risks faced by LPs on AMMs as discussed in Section IV. After the events expire, the module tokens can be redeemed for binary events with $False$ state as well, *i.e.,* $\forall i$ s.t. $s_i = False$, P_i and U_i can be redeemed for:

$$P_i : \epsilon \cdot C, \quad U_i : (1 - \epsilon) \cdot C \tag{2}$$

Note that the above payoff values satisfy **Invariant** (2) since the sum of payoffs of $1 \cdot P_i$ and $1 \cdot U_i$ is always $1 \cdot C$, regardless of the event's state. The redemptions are implemented using the `RedeemP(i)` and `RedeemU(i)` methods, respectively, which are depicted in the middle and right parts of Figure 1.

User Incentive Justification: Let p_{P_i}, p_{U_i} be the prices of P_i, U_i in terms of C. Then, a policyholder purchasing $1 \cdot P_i$ from the market is effectively paying a one-time premium of p_{P_i} to obtain an insurance coverage of $(1 - \epsilon) \cdot C$. This gives a coverage-to-premium ratio of the policy to be $(1 - \epsilon)/p_{P_i}$.

On the other hand, a user will underwrite a policy only if there is sufficient incentive for them. One such scenario is when the price of the policy token P_i increases significantly. This is because **Invariants** (1),(2) imply:

$$1 \leq p_{P_i} + p_{U_i} \leq 1 + f \tag{3}$$

as one can always acquire $1 \cdot P_i + 1 \cdot U_i$ for $(1+f) \cdot C$ and dispose the same for $1 \cdot C$ directly from the core contract using `Mint()` and `Burn()` respectively. Therefore, if the market price of the policy token increases by more than $f + \epsilon$, the following holds:

$$f + \epsilon + p_{U_i} < p_{P_i} + p_{U_i} \leq 1 + f \tag{4}$$

Hence, the price of the underwriting token becomes less than $1 - \epsilon$. This incentivizes underwriters to acquire U_i at a lower price from the market and later redeem it using `RedeemU()` for $(1-\epsilon) \cdot C$ if the assertion does not hold, with the difference $(1-\epsilon) \cdot C - p_{U_i}$ representing the earned premium. In summary, the immutability of the core module and an invariant-based redemption guarantee zero risk of default, thereby ensuring the robustness of the BAKUP protocol.

B. Oracle Module

The oracle module is an externally deployed contract with a custom-defined logic. This logic takes input values from external data sources and executes the logic of the assertion for each event. A user willing to create a core contract on BAKUP specifies an oracle contract address to associate with it. This is because the core module consists of a `Trigger(i)` function for the i^{th} event that executes a callback function `OTrigger(i)` in the oracle contract. `OTrigger(i)` executes the logic of the i^{th} assertion and returns a boolean value back to `Trigger(i)` in the core contract. This is presented as a flowchart diagram in Figure 2.

The returned value $False$ is ignored; however, in the other case, the following occurs:

1) s_i transitions to *True* and becomes immutable.
2) The `RedeemP(i)`, and `RedeemU(i)` methods are set to exchange as per (1).

When the contract deadline passes, the following occurs:

1) The `Trigger()` method is rendered invalid.
2) s_i becomes immutable for all events.
3) The `RedeemP(i)` and `RedeemU(i)` functions become valid for events with a $False$ state, redeeming tokens based on Relation (2).

Custom developer-defined logic enables extensive functionalities. Simultaneously, a modular structure ensures that the accounting performed by the core module remains unaffected by the execution and vulnerabilities of the oracle module. This allows users to reconcile platform resilience and policy flexibility simultaneously.

C. Yield Module

The BAKUP protocol enables the deployment of custom peripheral yield modules where P- and U-holders have the *option* to earn yield on their locked C. The key idea is to utilize the perpetual nature of the `Mint()` and `Burn()` methods of the core module, along with the trustless execution of smart contracts. Below, we present methods for a module design that delegates yield generation to a third-party protocol Y, earning yield on C. Figure 3 depicts the sequence of the following methods.

1) `Deposit()`**:** P- and U-holders willing to earn yield on platform Y deposit their tokens in the yield module. Let there be k_P units of P and k_U units of U and let us define λ as $min(k_P, k_U)$.
2) `Invest()`**:** Since one can only burn equal quantities of the two tokens, the yield module burns λ units of P and U to obtain λ units of C and invests this capital on platform Y. Observe that for maximum utilization of the tokens, the two tokens need to be equal.
3) `Divest()`**:** Here, we assume that yield is earned for a fixed duration of time. After its expiration, the module divests its position from platform Y to obtain λ' units of C. Generally, λ' is greater than λ due to accrued interest. However, there is also the scenario where the risk exposure of Y leads to an overall loss. In such a case, λ' is less than or equal to λ.
4) `Distribute()`**:** The yield contract mints $\lambda'/(1+f)$ units of P and U and distributes them, along with the unburnt tokens, uniformly among depositors.

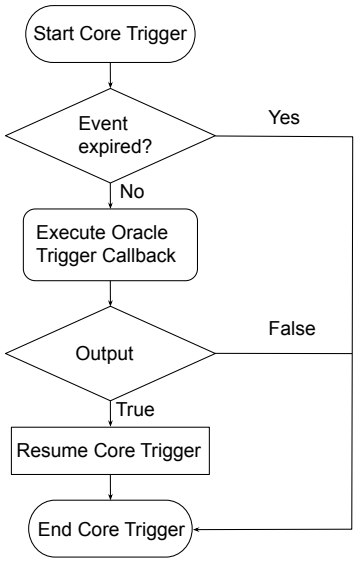


Fig. 2: Flow of Trigger execution using Oracle module.

Without loss of generality, let $k_P \geq k_U$ making $\lambda = k_U$ and let $k'_U = \lambda'/(1+f)$. Then, in the scenario of a successful (profitable) divesting, $k'_U - k_U$ units of P and U are uniformly distributed as yield to their holders. Thus, the holder of $1 \cdot U$ receives a yield fraction of $(k'_U - k_U)/k_U$ while the holder of $1 \cdot P$ receives a yield fraction of $(k'_U - k_U)/k_P$, which is k_U/k_P times the yield earned per unit of U. Therefore, if there are more policyholders than underwriters, k_U/k_P becomes smaller, attracting fewer additional policyholders than underwriters until k_U/k_P returns to 1. This serves as a negative feedback mechanism that tends to maintain an equal number of P and U depositors.

Since the yield module operates at the periphery, it does not affect the logic of the core module. Thus, a user investing in a different platform Y' has an independent risk exposure from the above user. Therefore, such a design strategy allows users to manage their risk individually.

IV. DIVERGENCE LOSS EVALUATION

Divergence Loss is defined as the opportunity cost for an LP to provide token reserves as liquidity compared to just holding them. In this section, we assess the divergence loss of LPs in uniform liquidity provision. Here, we assume an AMM pool of the pair P_i/U_i and that an LP mints 1 unit of P_i and U_i and creates a liquidity position with an entry price $p_i = \mathbf{p}$. If

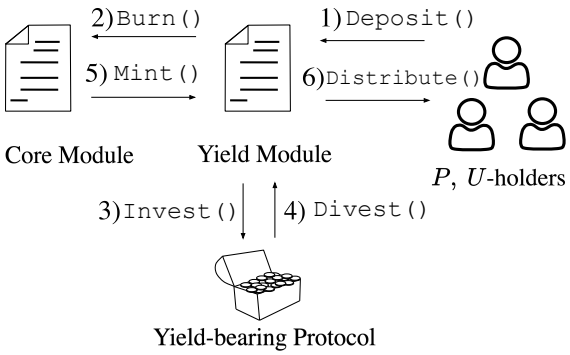
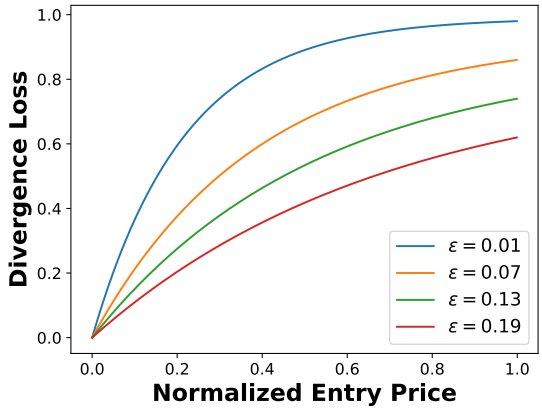
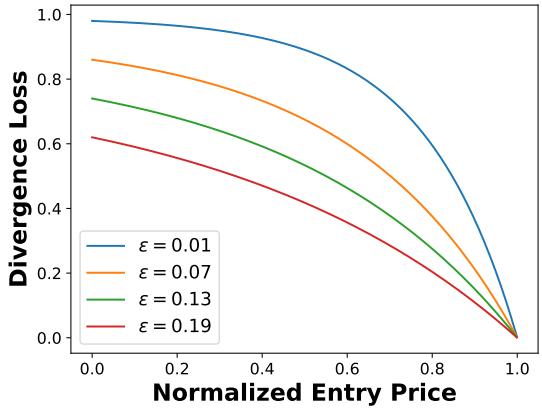


Fig. 3: Sequence of methods in the yield module with arrows indicating direction of token flow.



(a) No catastrophe scenario



(b) Catastrophic scenario

Fig. 4: Divergence loss in uniform liquidity provision in scenarios without and with catastrophes.

there is no liquidity provision, the value of these tokens at the conclusion of the contract is always 1 unit of C (enforced by Invariant (2)). To evaluate the divergence loss, we calculate the total value of the tokens held by the LP at the conclusion of the contract and represent it as a fraction of $1 \cdot C$. We use four equally-spaced values for $\epsilon : \{0.01, 0.07, 0.13, 0.19\}$ and do not consider any further higher values since $\epsilon \ll 1$. For plotting, we take the natural logarithm of the x-axis, for the price interval $[\frac{\epsilon}{1-\epsilon}, \frac{1-\epsilon}{\epsilon}]$ (extreme of possible prices at conclusion), and then min-max normalize it to the range $[0, 1]$.

If there is no catastrophe, *i.e.,* the underlying assertion does not hold, the LP ends up with only P_i. Their $1 \cdot U_i$ is exchanged for an average price of $\sqrt{\frac{\epsilon \mathbf{p}}{1-\epsilon}}$. Therefore, the LP ends up with $1 + \sqrt{\frac{1-\epsilon}{\epsilon \mathbf{p}}}$ units of P_i each worth $\epsilon \cdot C$, thus a total value of $\epsilon + \sqrt{\frac{\epsilon(1-\epsilon)}{\mathbf{p}}}$ units of C.

When there is a catastrophe, the LP ends up with only U_i. As above, their $1 \cdot P_i$ is exchanged for an average price of $\sqrt{\frac{(1-\epsilon)\mathbf{p}}{\epsilon}}$, giving them a total of $1 + \sqrt{\frac{(1-\epsilon)\mathbf{p}}{\epsilon}}$ units of U_i. Since U_i is worth $\epsilon \cdot C$, they receive a value of $\epsilon + \sqrt{\mathbf{p}\epsilon(1-\epsilon)}$ units of C.

The above two cases are shown in Figure 4a & 4b, respectively. We can observe that when the normalized entry price is closer to $0(1)$, the divergence loss is higher for the catastrophic(non-catastrophic) scenario. Also, for a given $\mathbf{p}$, the divergence loss is lower at higher values of ϵ. The worst loss (at $\mathbf{p}$=$\frac{\epsilon}{1-\epsilon}$ or $\frac{1-\epsilon}{\epsilon}$) can be reduced from 0.98 to 0.62 by varying ϵ from 0.01 to 0.19. Lastly, if $\epsilon = 0$, LPs incur a divergence loss of 100% underscoring the lower bound on ϵ.

V. Related Work

There have been several designs proposed for smart contract-based decentralized insurance for DeFi. Nexus Mutual [13] and inSure [14] are the largest of them and are inspired by the design of mutual insurance [15]. Unlike BAKUP, they do not support permissionless policy listing or optional yield generation. Rather, a fraction of the pooled capital is invested in yield-generation protocols, transferring risk to all members. These decisions and others, like claim assessment, are done via voting using governance tokens.

Other protocols, including ours, are based on Peer-to-Peer decentralized insurance [16] where individuals pool their insurance premiums and use these funds to mitigate individual damages. Some of them including Risk Harbour [17], Opium Protocol [18], cozy.finance [19], and Etherisc [20] facilitate automated underwriting done via smart contract while others like Unslashed Finance [21], Nsure [22], and Cover [23] have a voting-based claim assessment. Moreover, some of them including [18], [19] allow permissionless listing. In contrast to BAKUP, none of the above protocols give their users the optional ability to manage yield on their capital. Instead, either the protocol offers no yield, or the yield management is performed by governance or delegated to a third party [24]. On the other hand, BAKUP disjoints the core, oracle, and yield modules and allows users to manage yield on their staked capital while having a permissionless policy creation.

VI. Conclusion

A robust risk management system for the emerging area of DeFi requires a primitive platform (building block) with minimal external dependencies. This is successfully achieved in the BAKUP protocol presented in this paper through a modular approach. At the same time, the incentive alignment between participants, including module creators, underwriters, policyholders, and yield bearers, is studied. Possible future work in this direction includes studying the incentives and risks of liquidity providers provisioning liquidity for the conditional tokens on AMM, as well as extending the conditional tokens from binary to n-ary with n states to reduce the number of ERC20 tokens per adversity.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, https://bitcoin.org/bitcoin.pdf.

[2] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014, https://ethereum.org/en/whitepaper/.

[3] T. Rocket, "Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies," *Available [online].[Accessed: 4-12-2018]*, 2018.

[4] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0. 8.13," *Whitepaper*, 2018.

[5] A. Skidanov and I. Polosukhin, "Nightshade: Near Protocol Sharding Design," 2019, https://near.org/downloads/Nightshade.pdf.

[6] H. Arslanian, "Decentralised finance (defi)," in *The Book of Crypto: The Complete Guide to Understanding Bitcoin, Cryptocurrencies and Digital Assets*. Springer, 2022, pp. 291–313.

[7] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 2017, pp. 164–186.

[8] L. Gudgeon, D. Perez, D. Harz, B. Livshits, and A. Gervais, "The decentralized financial crisis," in *2020 crypto valley conference on blockchain technology (CVCBT)*. IEEE, 2020, pp. 1–15.

[9] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2444–2461.

[10] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022, pp. 30–46.

[11] F. Bekemeier, "Deceptive assurance? a conceptual view on systemic risk in decentralized finance (defi)," in *Proceedings of the 2021 4th International Conference on Blockchain Technology and Applications*, 2021, pp. 76–87.

[12] "Lido: Ethereum liquid staking," 2020, https://lido.fi/static/Lido:Ethereum-Liquid-Staking.pdf.

[13] H. Karp and R. Melbardis, "Nexus mutual whitepaper: A peer-to-peer discretionary mutual on the ethereum blockchain," 2017, https://nexusmutual.io/assets/docs/nmx_white_paperv2_3.pdf.

[14] inSureDAO, "insure ecosystem: Decentralized insurance platform," https://insuretoken.net/whitepaper.html.

[15] P. Albrecht and M. Huggenberger, "The fundamental theorem of mutual insurance," *Insurance: Mathematics and Economics*, vol. 75, pp. 180–188, 2017.

[16] R. Feng, M. Liu, and N. Zhang, "A unified theory of decentralized insurance," *Available at SSRN 4013729*, 2022.

[17] M. Resnick, R. Ben-Har, D. Patel, and A. Bipin, "Risk harbor v2," 2022, https://github.com/Risk-Harbor/RiskHarbor-Whitepaper/blob/main/Risk%20Harbor%20Core%20V2%20Whitepaper.pdf.

[18] O. team, "Opium protocol whitepaper," 2020, https://github.com/OpiumProtocol/opium-contracts/blob/master/docs/opium_whitepaper.pdf.

[19] Cozy.Finance, "Cozy finance developer docs," 2020, https://docs.cozy.finance/.

[20] Etherisc, "Whitepaper," 2022, https://docs.etherisc.com/learn/whitepaper-en.

[21] Unslashed.Finance, "Insurance for decentralized finance," 2021, https://documentation.unslashed.finance/.

[22] Nsure.Network, "Open insurance platform for open finance," 2020, https://nsure.network/Nsure_WP_0.7.pdf.

[23] "Cover protocol," https://github.com/CoverProtocol.

[24] "Enzyme finance," https://docs.enzyme.finance/.