Dispersing Asymmetric DDoS Attacks with SplitStack

Ang Chen[†][*] Akshay Sriraman[†] Tavish Vaidya[‡] Yuankai Zhang[‡]
Andreas Haeberlen[†] Boon Thau Loo[†] Linh Thi Xuan Phan[†]
Micah Sherr[‡] Clay Shields[‡] Wenchao Zhou[‡]
[†] University of Pennsylvania [‡] Georgetown University

ABSTRACT

This paper presents SplitStack, an architecture targeted at mitigating asymmetric DDoS attacks. These attacks are particularly challenging, since attackers can use a limited amount of resources to trigger exhaustion of a particular type of system resource on the server side. SplitStack resolves this by splitting the monolithic stack into many separable components called minimum splittable units (MSUs). If part of the application stack is experiencing a DDoS attack, SplitStack massively *replicates just the affected MSUs*, potentially across many machines. This allows scaling of the impacted resource separately from the rest of the application stack, so that resources can be precisely added where needed to combat the attack. We validate SplitStack via a preliminary case study, and show that it outperforms naïve replication in defending against asymmetric attacks.

Categories and Subject Descriptors

G.2.0 [**General**]: Security and protection

Keywords

Denial-of-service attacks, denial-of-service defenses

1. INTRODUCTION

Stopping denial-of-service attacks on networked services remains a challenging problem, and stories about successful attacks on commercial services [36, 43] or the Internet's infrastructure [44] are still disturbingly common in the news. Existing solutions primarily focus on stopping the attack traffic as early as possible. This can be done at the source by blocking transmissions on compromised machines [23]; it can be done in the network by filtering traffic at routers [31]; or it can be done at the end hosts by trying to recognize and block bogus requests before they can consume any resources [26, 30]. None of these techniques are perfect. Hence, it remains important to harden a data center against attacks that manage to pass through all the other defenses.

We characterize two kinds of denial-of-service attacks that can reach a data center based on the resources involved in the attack. In a *symmetric attack*, the resources required from the attacker are of the same type and scale as those denied the victim. For example, a network flooding attack requires the attacker to use significant network bandwidth; the same amount of bandwidth is consumed at the victim who receives it. In this case, the defender can succeed by matching the attacker's resources using over-provisioning and/or massive replication. Such attacks have been studied in detail, and there are commercial solution – e.g., from major CDNs [5] – that can provide the necessary resources on demand.

Defending against an *asymmetric attack* is more difficult. In these attacks, the resources used by the attacker and the victim resources in the data center differ in type or scale, or both. Common examples include those in which attacker traffic consumes limited computational resources or requires maintenance of state in finite memory. For instance, in TLS renegotiation attacks [2], the attacker constantly asks the servers to generate new key material for an existing TLS connection, which is computationally expensive. As a result, the servers' CPUs are loaded with cryptographic operations and cannot handle requests from legitimate clients, even though all the other necessary resources (memory, network bandwidth, I/O bandwidth, etc.) remain available. There is a variety of known asymmetric attacks [24], including SlowPOST/Slowloris attacks [37], zero-length TCP window attacks [13], and ReDoS attacks [1], etc. These attacks are trickier to handle because they can succeed even if the attacker has vastly fewer resources than the defender. Our goal is to develop a new kind of defense against those attacks.

There are two fundamental reasons why asymmetric attacks are so difficult to defend against. The first is their great *diversity*: in contrast to symmetric attacks, which all tend to use the same brute-force approach, each asymmetric attack targets one particular weakness in the defender's service, such as the complexity of Regex parsing in ReDoS, or the server's limited connection pool in Slowloris. As a result, although defenses exist in the market for dealing with these attacks, they tend to be specialized point solutions – for instance, a defense against ReDoS attacks would be useless against Slowloris attacks, and vice versa. Therefore, it is difficult to get a truly comprehensive defense against asymmet-

[*] The authors are listed alphabetically, with student authors appearing before faculty authors.

HotNets-XV, November 9–10, 2016, Atlanta, GA, USA.
Copyright is held by the authors. Publication rights licensed to ACM.
ACM 978-1-4503-4661-0/16/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/3005745.3005773.

Attack	Target resource	Existing defenses
SYN-flood [17]	Half-open connection pool	SYN cookies
TLS renegotiation [2]	CPU cycles spent on TLS handshakes	SSL accelerators
ReDOS [1]	CPU cycles spent on Regex parsing	Regex validation
SlowPOST/Slowloris [37]	Established connection pool	Increase connection pool size
HTTP GET flood [12]	CPU cycles and memory	Rate limiting
Christmas tree attack [7]	CPU cycles spent on processing packet options	Filtering
Zero-length TCP window [13]	Established connection pool	Increase connection pool size
HashDoS [8]	CPU cycles spent on maintaining hash tables	Use stronger hash functions
Apache Killer [33]	Memory	Allocate more memory

Table 1: Examples of asymmetric denial-of-service attacks.

ric attacks: even if the defender deploys all known defenses in combination, the result is still unlikely to help against an attack that uses a new attack vector.

The second reason is that today's application stacks are *monolithic*, so they place constraints on how available resources can be used. Consider, for instance, a TLS renegotiation attack on a typical two-tiered web service that consists of a HTTP server tier and a database tier. Since TLS is handled exclusively by the HTTP servers, the attacker can win by exhausting the CPU resources on these servers – even if the database servers are completely idle, and have lots of CPU resources that could have been used to help the first tier! We could increase the CPU resources by spinning up more nodes as HTTP servers, but this ties down all other resources on these nodes (e.g., memory, I/O bandwidth) that could be used elsewhere, and they essentially go to waste. Notice that *this limitation exists only because of the monolithic architecture in current HTTP servers*. If we could find a way to carve out the TLS handshake component, and replicate just this component on the database tier to use its idle CPU cycles, it would be a much more effective defense.

In this paper, we propose a new approach to handling asymmetric denial-of-service attacks, with three key elements. First, we propose to break up the application stack into smaller components that can be moved and replicated independently. This is inspired by the current trend towards micro-services [40], but our vision goes much further: we aim to operate at a much smaller granularity, e.g., by moving the TLS handshake component or the SYN processing component. Second, we propose to add a centralized controller that assigns components to machines and routes data flows between them, much like an SDN controller routes packet flows between switches. This controller could carefully schedule the components, e.g., to meet a given SLA objective. Third, we propose to continuously monitor the resource consumption of each component, and to replicate any components that are overloaded, e.g., due to a denial-of-service attack. We refer to this design as the *SplitStack architecture*.

The SplitStack architecture offers two benefits for defending against asymmetric attacks. First, the fine-grained components make it easier for the defender to deploy all available resources on all machines against the attacker, exactly as needed. For instance, SplitStack could respond to a TLS renegotiation attack by temporarily enlisting the database tier, or even machines from other services, to help with TLS handshakes. Second, the reactive replication approach is not attack-specific and can thus potentially mitigate unknown asymmetric attacks. Once SplitStack recognizes that a component is overloaded or its throughput appears to drop, it can respond by replicating that particular component – without having seen the attack before, and without knowing the specific vulnerability that the attacker is targeting! This is especially useful because DDoS attacks today tend to use multiple attack vectors [27].

We do not intend SplitStack as a cure against all possible DDoS attacks. If an attacker can fill up the data center's inbound link or completely overwhelm the defender's resources, she can still succeed. We also do not envision SplitStack as the *only* defense against attacks: specialized defenses, such as hardware SSL accelerators [18], can be more efficient than SplitStack because they are tailored to a particular attack vector. Rather, we propose SplitStack as a generic defense that can mitigate an attack (particularly unknown attacks) at least until help arrives. As a welcome side-effect, SplitStack's fine-grained scheduling and migration techniques provide more freedom for matching up tasks and resources and could thus increase utilization in data centers and/or provide better QoS even in the absence of attacks.

As a proof-of-concept, we conducted an initial set of experiments with a preliminary SplitStack prototype to show that SplitStack's approach can be effective, and that it can utilize all available resources when under attack. Looking forward, there is clearly a lot of work left to be done. For instance, it would be useful to have an automated way to split existing network stacks into fine-grained components (perhaps with a bit of help from the developer), and there may be cases where a 'component' cannot be split and replicated easily (e.g., when consistency requirements are involved). On the other hand, an architecture like SplitStack could also provide many interesting new opportunities: by separating out a 'control plane' for software systems, analogous to the control plane in software-defined networks, SplitStack could potentially enable entirely new kinds of policies, e.g., for scheduling, migration, and fault-tolerance.

2. OVERVIEW

As a running example, we consider a two-tiered web service hosted in a data center, where the frontend is an HTTP server tier, and the backend is a database tier. An external adversary launches a TLS renegotiation attack [2] that consumes CPU resources on the HTTP servers, by constantly asking

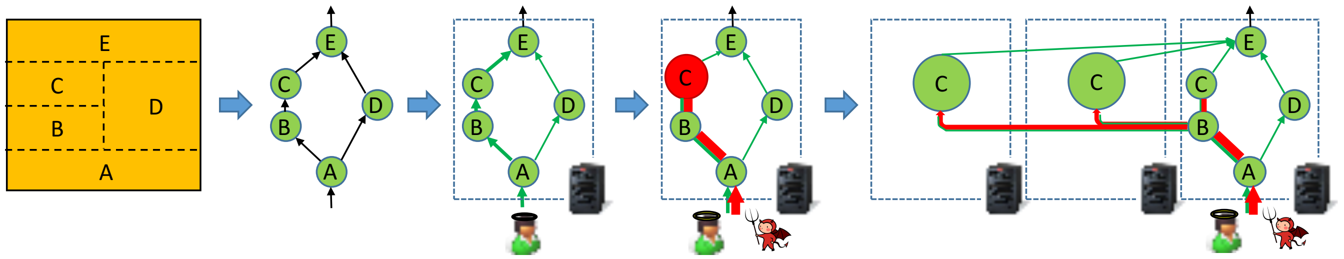

Figure 1: Example use case of SplitStack. The monolithic software (a) is transformed into a dataflow graph (b) with smaller components, called MSUs, which are then scheduled on the available machines (c). When an attacker attempts to overload one of the components (d), SplitStack disperses the attack by generating additional instances on other machines (e).

the servers to switch to a new cryptographic key. As a result, legitimate requests are being served very slowly, or not at all. (Of course, SplitStack is applicable to other types of asymmetric attacks as well, e.g., those shown in Table 1.)

Threat model: In *asymmetric DDoS attacks*, the attacker is unable to match the defender's bandwidth, but she succeeds by cleverly "choking" some other resources, e.g., CPU cycles, memory. We do not assume that the attack vector is known to the operator – novel attacks are explicitly in scope. Our goal is to *automatically mitigate an attack, even if it has a new attack vector*, and to provide reasonably good quality of service to the legitimate clients while the human operators are working on a more permanent solution. We *do not* consider brute-force attacks that saturate a data center's ingress link, or exploits that take over data center machines.

2.1 Strawman Solutions

One strawman defense is to *filter or block* suspicious network traffic. For instance, if the service is receiving a huge number of `GET /redsox.mov` requests, the operators can simply block access to `/redsox.mov`. However, this heavily relies on the accuracy of request classification, so it is susceptible to false positives and negatives – what if the requests are really coming from baseball fans after a successful game by the Boston Red Sox? Also, an adversary can send a heterogeneous mix of requests to confuse such a defense.

Another approach is to increase capacity by *replication*. For instance, to handle a TLS renegotiation attack, an operator can launch more web server nodes to increase the number of TLS connections that can be handled. This defense does not critically depend on an accurate classification, but it is very inefficient: every new machine will contribute a bit more CPU power, while its other resources (memory, bandwidth, I/O, ...) will be heavily underutilized or go to waste.

2.2 Approach: SplitStack

Our observation is that the data center often has a lot more of the overloaded resources elsewhere, but the current software architecture cannot effectively use them. For instance, in our example, the CPUs on the database servers will be mostly idle while the web servers struggle to keep up with the attack. If the database servers were able to "help" the web servers by contributing their computation power, the capacity at the bottleneck (TLS renegotiation) would increase.

We propose such "helping" by splitting the monolithic application stack into smaller pieces that can be replicated and migrated independently. This additional flexibility would enable an attacked service to use *all* of the available resources for its defense – by temporarily enlisting other servers, or even machines from different services. The effect would be a substantial increase in the service's capacity, and thus, hopefully, better quality of service for the legitimate clients.

2.3 Challenges

This approach raises a number of important questions. First, *where and how should we split* a monolithic software into smaller pieces? If the splitting is not done carefully, SplitStack may not be able to provide correctness guarantees. Second, *how should the pieces coordinate* with each other to best utilize the available resources? Finally, *who should make the decision to split* and move the pieces to find the best way to respond to a given attack?

3. THE SPLITSTACK ARCHITECTURE

The SplitStack architecture models a monolithic application stack as a dataflow graph consisting of Minimum Splittable Units (MSUs), each of which can be further annotated with an expected execution time and deadline (if any). MSUs are deployed in lightweight containers, and can be replicated on one or more machines.

Figure 1 shows how this approach can disperse a DDoS attack. The monolithic software (a) is transformed into a dataflow graph with several MSUs (b), which are then scheduled by a controller to run on one or more available machines so as to meet specified deadlines (c). When the system comes under attack, one of the MSUs becomes overloaded and prevents the software from handling legitimate requests (d). The controller detects this based on the gathered performance statistics, automatically creates additional instances of the affected MSU on other machines, and balances the workload across them (e). Meanwhile, SplitStack alerts the operator and provides diagnostic information, so that she can better understand the attack vector, e.g., a bug in the affected MSU, and find a long-term solution. Thus, as long as the system *as a whole* has enough resources, all of the incoming requests can still be handled.

3.1 Minimum Splittable Units

In Figure 1(b), each vertex in the dataflow graph represents an MSU. An MSU is a small, (mostly) self-contained functional unit with narrow interfaces to other MSUs. It contains four types of meta data: a) a primary key to unique identify an MSU, b) a routing table that steers requests to next-hop MSUs, c) a cost model, which we describe more in Section 3.4, and d) typing information, which specifies how an MSU communicates with its replicas after being cloned into multiple copies (certain kinds of MSU replicas can operate independently; other kinds would need to coordinate).

Inter-MSU communication takes place via IPC when the MSUs are located on the same node (or even via function calls when they are located in the same address space), but it can be transparently switched to RPCs after an MSU migration. This is because the SplitStack controller may transform the dataflow graph in response to an attack, invoking four transformation operators on MSUs: `add`, `remove`, `clone`, and `reassign`. The MSUs and transformation operators form a basis for a SplitStack to defend against DDoS attacks.

3.2 Software Partitioning

Partitioning a monolithic software into MSUs requires an intricate balance: if an MSU contains too little functionality (e.g., wrapping each function into its own MSU), it may need to constantly coordinate with other MSUs to get things done, resulting in high overhead; if an MSU is too large, then we cannot easily achieve the fine-grained responses we desire. Therefore, one rule of thumb for partitioning MSUs is that the cost incurred by book-keeping and communications between MSUs should be much less than the cost of replicating a larger component in the software.

Fortunately, the layered nature of the network stack provides a useful starting point. The cross-layer interfaces are already reasonably narrow for use as a first approximation of MSU boundaries. For each layer, we further adopt a static partitioning of MSUs, including TCP handshake MSUs, TLS negotiation MSUs, etc., somewhat analogous to what Click [28] has done. As past work – e.g., on SawMill Linux [21] – has shown, manually splitting complex software, such as an OS kernel, is feasible but difficult. We are developing ways to automate this process in our ongoing work (see Section 6).

3.3 MSU Coordination

At runtime, additional MSU instances may be replicated and migrated, so we need to carefully coordinate the state across MSU instances. SplitStack achieves this by applying different coordination mechanisms depending on the functionality of the MSU, and on how the MSU handles requests.

Independent MSUs: Some MSUs can process each request in isolation. For instance, our current SplitStack prototype has a TCP handshake MSU that can serialize, marshal, and migrate a completed TCP connection to its downstream application-layer MSUs, using the TCP connection repair functionality [15] available in Linux v3.5 and above. It also has a TLS negotiation MSU; transferring state from this MSU to its downstream MSUs is also just a matter of migrating the appropriate keys, secrets, and ciphersuite selections. In both cases, the MSU replicas can operate independently.

In the case of such "siloed" MSUs, the `reassign` and `clone` SplitStack operations are simple: reassign transfers the corresponding state to the new instance, and clone can be performed without any coordination whatsoever. Migrating state from one MSU to another (i.e., during `reassign`) could be performed either as an offline or live process. In the offline case, SplitStack reserves resources to construct the new MSU, the existing MSU is stopped, state is transferred, and the new reassigned MSU is then activated. Under load, such offline migration may be too costly since transferring state could be slow, thus incurring an unacceptable downtime. Inspired by techniques for live VM migration [14], SplitStack uses iterative copy and commitment phases that more slowly migrate state while allowing the existing MSU to service requests until the new MSU is activated. Live migration minimizes downtime at the expense of a longer overall `reassign` operation.

Handling state: Coordinating state is more difficult for MSUs with cross-request dependencies. For example, an MSU that handles the search function of a web application may require access to state related to the user's access permissions and prior requests, both of which may be affected by separate MSUs. A simple approach is to maintain and access such state only through a centralized memory store such as Redis. (This model is already becoming widely adopted for applications deployed as a collection of microservices.) In Section 6, we sketch a potential solution for coordinating cross-request state between MSUs in a distributed fashion.

Routing requests through MSUs: As SplitStack dynamically schedules MSUs on multiple physical nodes, control and data traffic is routed accordingly to ensure that requests arrive at the correct MSUs, using a "routing table" in each MSU. For example, when multiple MSUs are created to scale the processing of a particular functionality (such as handling TLS key negotiation), the incoming traffic is divided evenly among these MSUs. SplitStack preserves flow affinity requirements for MSUs whenever appropriate.

3.4 The SplitStack Controller

SplitStack has a central controller that is responsible for allocating resources and scheduling the MSU graph at runtime. Scheduling decisions include the initial and subsequent placement of the (instances of the) MSUs on the machines, the scaling of the MSUs in response to potential attacks, and the assignment of requests to MSU instances. The controller's goal is to balance loads across the data center to meet a certain SLA. By default, our scheduler uses the standard Earliest Deadline First (EDF) algorithm within each node for predictable performance.

SplitStack accepts an overall SLA requirement for an application in the form of end-to-end latency constraints. In the software partitioning phase (Section 3.2), SplitStack obtains the MSU-level deadlines by dividing the end-to-end latency constraint among the MSUs along a path of the graph, proportionally to their computation costs.

Cost model. To make resource allocation decisions, the SplitStack controller needs to know the execution requirements of each MSU, in the form of its *cost model*. Concretely, the cost model for each MSU includes (a) the amount of computation resource needed to process an input data item (e.g., a packet or an RPC), (b) the number of output data items to be transmitted to a downstream MSU, and the amount of network bandwidth required for each item, and (c) the effect of the graph operators on the MSU. Since these resource requirements can change drastically at runtime, e.g., during algorithmic complexity attacks, SplitStack periodically updates the cost model based on the monitoring information gathered at runtime. When not provided by the operator, the computation overhead (i.e., the worst-case execution time, or WCET [9]) of each MSU and the communication costs among them can be estimated using either static analysis of the source code (e.g., using existing timing analysis tools developed in the real-time community [9]) or profiling (if only binaries are available).

MSU placement. Based on the cost models and individual MSU deadlines, the SplitStack controller formulates the initial placement of MSUs on machines and the assignment of requests to the MSU instances as an optimization problem. It uses two kinds of constraints: (a) the *total utilization* of the MSUs on each core should be at most one, to ensure that MSUs meet their deadlines; and (b) the resulting *total bandwidth required* on each network link for the communication among MSUs on different machines should not exceed the link's available bandwidth. The optimization objective aims to, first, minimize the worst-case bandwidth requirement on a network link, and then minimize the worst-case CPU utilization per machine, so as to balance load across the machines and network links. When possible, MSUs that are adjacent in the dataflow graph are scheduled on the same machine, so that they can communicate using IPC (or even function calls!) rather than using network messages.

Monitoring and adaptation. At runtime, the controller detects bottlenecks by monitoring the system, using a set of monitoring agents on each machine. The data is aggregated hierarchically reduce communication overhead. The agents keep track a range of critical metrics necessary for the detection of potential DDoS attack, including the fill levels of the input and output queues, the current CPU load, memory and I/O utilization on each machine, and the load at each router. SplitStack reserves a fixed amount of the available bandwidth for the communication between the monitoring component and the controller.

When a potential DDoS attack is detected, the controller invokes the four transformation operators to scale the MSUs, re-allocate resources, re-assign requests, and update the routing tables and cost models for the MSUs. Our initial SplitStack controller uses a greedy approach – it assigns cloned MSU instances based on the least utilized machines and network links, while ensuring the two utilization and bandwidth constraints are satisfied – but we are currently working on more refined strategies. The controller also periodically rebalances the load among the data center resources by resolving the optimization problem with updated information, while minimizing changes to the current allocation.

If the controller blindly replicated overloaded MSUs on random nodes, it could take resources away from other services and/or consume additional bandwidth for inter-MSU communication, which could further aggravate the situation. Hence, it is essential for the controller to have a global view and to find solutions that provide good overall performance.

4. CASE STUDY

To validate that SplitStack can disperse asymmetric DDoS attacks more effectively than naïve replication, we have developed a proof-of-concept prototype of SplitStack, and we have conducted a set of experiments on five DETERLab [6] nodes. Our server-side setup consisted of one ingress node, and three service nodes; all incoming requests arrive at the ingress and then get processed on the service nodes. Initially, only two service nodes were activated – one node ran an Apache v2.4 web server, and another ran a MySQL v5.7.12 database; the web server was backed up by the database using a PHP v7.0 framework. In the absence of attacks, the third service node was idle. The attacker resided on a fifth DETER node that was connected to the ingress.

In our experiment, the attacker launched a TLS renegotiation attack using the `thc-ssl-dos` [39] tool, which exhausted the computation resources on the web server node with frequent TLS renegotiations. To defend against this, a naïve replication strawman approach replicated one additional web server on the idle service node, and balanced the incoming requests between the two web servers using HAProxy v1.6.3. SplitStack, in contrast, recognized the *TLS handshake component* as the MSU, and replicated only this MSU. SplitStack enlisted not only the idle service node, but also the database node and the ingress node; we used this as a first approximation of the kind of fine-grained replication strategy SplitStack enables. Therefore, SplitStack replicated three additional components on these three nodes; the incoming requests were then balanced among the four components for the TLS handshaking stage. This was approximated by launching three `stunnel` [42] v5.3.4 proxies on those nodes which help with TLS processing, which then handed off established connections to the Apache web server.

Note that we are able to create three additional `stunnel` instances in SplitStack, as opposed to only one additional instance of web server in the naïve approach, because the `stunnel` component is a lightweight process with comparatively smaller memory and computational footprint. Consequently, SplitStack was able to utilize spare cycles on the database and ingress nodes for running an extra instance of `stunnel`, which the naïve approach was unable to do.

Figure 2 shows the comparison between the three different responses to the DDoS attack: (a) the "no defense" approach

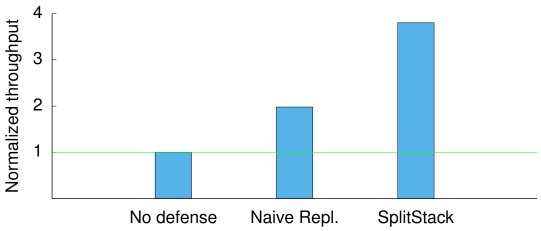

Figure 2: Comparison of three defense mechanisms.

deployed no additional replication, (b) the "naïve replication" approach instantiated one additional web server, and (c) the SplitStack approach replicated three impacted MSUs. For each setup, we measured the maximum number of attack handshakes the web service can handle per second. We observe that naïve replication does help with alleviating the attack – the web service can handle about 1.98 times as many handshakes per second. SplitStack, on the other hand, can handle 3.77 times as many handshakes per second; this did not achieve a 4-times scale up, as one might expect, because the ingress node spent quite some CPU cycles on load-balancing the requests. Nevertheless, we can see that SplitStack still squeezed out significant computation power from the *remaining* CPU cycles. Overall, SplitStack achieved almost twice the throughput of naïve replication.

In practice, the improvement relative to naïve replication depends on the exact setup and could even be considerably higher than in our experiment. For instance, if we had a different number of additional nodes or VMs in the web service, the improvement ratio would change accordingly. Nevertheless, this initial set of results suggests that fine-grained replication, as enabled by SplitStack, can be a promising approach for defending against asymmetric DDoS attacks.

Our current prototype is not yet complete enough to allow a meaningful evaluation of SplitStack's overhead. For instance, the communication between MSUs can introduce delay or – if the MSUs are placed on different nodes – create additional traffic. We expect that a) the overhead will be low during normal operation, when MSUs will typically share an address space and "communicate" via function calls, much like the subsystems of a classical application do today, and that b) the overhead can be kept low even under attack, as long as the MSUs have narrow interfaces and the scheduler takes care to place related MSUs on the same node if it can. We are currently working on a full prototype that can be used to answer these questions experimentally.

5. RELATED WORK

Asymmetric DDoS attacks: There are many kinds of asymmetric DDoS attacks, including ReDoS [1] that uses malicious Regex patterns that take very long for servers to process, TLS renegotiation attacks [2] that exhaust CPU cycles with SSL/TLS requests, HTTP SlowLoris/SlowPOST [37] that send partial HTTP requests at a delayed rate in order to occupy server resources, HashDoS [8] that exploits weak hash functions to create hash collisions, just to name a few. Existing defenses create specialized solutions for each attack, so a solution for one attack almost never works for another. In contrast, SplitStack uses a single defense strategy for a wide variety of asymmetric attacks.

Dispersion-based defenses: Dispersing DDoS attacks can be achieved by load-balancing [35, 31, 29]. For instance, Ananta [35] proposes a software load-balancing architecture for high-speed data centers, Pushback [31] and CoDef [29] can rate-limit traffic to defend against DDoS attacks, etc. Dispersion can also be achieved by replicating the service under attack [25, 46, 19]. XenoService [46] replicates web sites across XenoService servers when they are under attack, [25] replicates an attacked service to a different location and confuses attacks by a random shuffling, Botahei [19] dynamically launches more VMs to defend against legacy attacks. SplitStack is related to both approaches, but it has two advantages: (a) SplitStack *only replicates impacted MSUs*, not an entire stack or service, so it outperforms naïve replication strategies; (b) SplitStack does not look for specific features of legacy attacks, so it *can handle new attacks* with an unknown attack vector.

6. DISCUSSION

We are currently exploring the following open problems.

Identification of split points. The effectiveness of SplitStack depends on a careful software partitioning. Our current design adopts a strawman approach that uses cross-layer interfaces and pre-defined software components as splitting points; however, there is a rich literature on program partitioning [41, 10, 34, 11, 47] that contains a variety of approaches for splitting monolithic programs into a set of communicating components, and that SplitStack can benefit from. We are also investigating the use of declarative networking approaches [22], which can provide component-based abstractions for building composable software modules, as well as control-flow graph analysis [32, 38, 20] and program slicing [45, 3] to partition legacy applications.

Coordinating inter-dependent MSUs. The current SplitStack only supports "siloed" MSUs – i.e., MSUs with no cross-request dependencies. In addition to exploring centralized solutions for cross-request state management, we are investigating the use of SDN features to *route* state information between MSUs involved in a user's requests. Our goal is to combine distributed shared memory systems such as Orbe [16] with SDN routing to ensure causal consistency [4] of cross-request information among MSUs.

Acknowledgments

We thank our shepherd Stefano Vissicchio and the reviewers for their helpful comments. This material is based upon work supported in part by the the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056, and NSF grants CNS-1054229, CNS-1453392, CNS-1527401, CNS-1513679, and CNS-1563873. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NSF.

7. REFERENCES

[1] Regular expression denial of service - ReDoS. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[2] SSL renegotiation DoS. https://www.ietf.org/mail-archive/web/tls/current/msg07553.html.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. PLDI*, June 1990.

[4] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[5] Akamai. Cloud security. https://www.akamai.com/us/en/cloud-security.jsp.

[6] T. Benzel. The science of cyber-security experimentation: The DETER project. In *Proc. ACSAC*, Dec. 2011.

[7] B. Brenner. TCP flag DDoS attack by Lizard Squad indicates DDoS tool development. https://blogs.akamai.com/2015/01/tcp-flag-ddos-attack-by-lizard-squad-indicates-ddos-tool-development.html.

[8] B. M. Carlson. A PoC hash complexity DoS against PHP. https://github.com/bk2204/php-hash-dos.

[9] S. Chattopadhyay, C. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified WCET analysis framework for multi-core platforms. In *Proc. RTAS*, 2012.

[10] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. SOSP*, 2007.

[11] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proc. EuroSys*, 2011.

[12] D. Cid. Layer 7 DDOS-blocking HTTP flood attacks. https://blog.sucuri.net/2014/02/layer-7-ddos-blocking-http-flood-attacks.html.

[13] Cisco. Microsoft windows TCP/IP connection exhaustion denial of service vulnerability. https://tools.cisco.com/security/center/viewAlert.x?alertId=18959.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. NSDI*, 2005.

[15] J. Corbet. TCP connection repair. https://lwn.net/Articles/495304/.

[16] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proc. SOCC*, 2013.

[17] W. M. Eddy. Defenses against TCP SYN flooding attacks. http://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-34/syn-flooding-attacks.html.

[18] F5. SSL Acceleration. https://f5.com/glossary/ssl-acceleration.

[19] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proc. USENIX Security*, Aug. 2015.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[21] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *Proc 9th ACM SIGOPS European Workshop*, pages 109–114, 2000.

[22] H. Gill, D. Lin, X. Han, C. Nguyen, T. Gill, and B. T. Loo. Scalanytics: A declarative multi-core platform for scalable composable traffic analytics. In *Proc. HPDC*, 2013.

[23] S. Guha, P. Francis, and N. Taft. ShutUp: End-to-end containment of unwanted traffic. Technical report, Cornell University, July 2008.

[24] D. W. Holmes. Defending against low-bandwidth, asymmetric denial-of-service attacks. Presentation at the RSA Conference Europe, Oct. 2013.

[25] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and W. Powell. Catch me if you can: A cloud-enabled DDoS defense. In *Proc. DSN*, June 2014.

[26] C. Jin, H. Wang, and K. G. Shin. Hop-count filtering: an effective defense against spoofed DDoS traffic. In *Proc. CCS*, 2003.

[27] C. Kern. Increased use of multi-vector DDoS attacks targeting companies. http://www.bsminfo.com/doc/increased-use-of-multi-vector-ddos-attacks-targeting-companies-0001.

[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[29] S. B. Lee, M. S. Kang, and V. D. Gligor. CoDef: Collaborative defense against large-scale link-flooding attacks. In *Proc. CoNEXT*, 2013.

[30] Q. Liao, D. A. Cieslak, A. D. Striegel, and N. V. Chawla. Using selective, short-term memory to improve resilience against DDoS exhaustion attacks. *Security and Communication Networks*, 1(4):287–299, 2008.

[31] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. In *Proc. CCR*, 2002.

[32] J. Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, June 2012.

[33] National Vulnerability Database. Vulnerability summary for CVE-2011-3192. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3192.

[34] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *Proc. NSDI*, 2009.

[35] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proc. SIGCOMM*, 2013.

[36] D. Pauli. Chinese gambling site served near record-breaking complex DDoS. July 2016. http://www.theregister.co.uk/AMP/2016/07/01/470_gbps_multivector_chinese_gambling.

[37] D. Senecal. Slow DoS on the rise. https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html.

[38] O. Shivers. Control flow analysis in Scheme. In *Proc. PLDI*, June 1988.

[39] The Hacker's Choice. The thc-ssl-dos tool. https://www.thc.org/thc-ssl-dos.

[40] Thoughtworks. Real-world microservices: Lessons from the frontline. 2014. https://www.thoughtworks.com/insights/blog/microservices-lessons-frontline.

[41] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1:1–1:40, Aug. 2009.

[42] M. Trojnara. The stunnel TLS proxy. https://www.stunnel.org/index.html.

[43] W. Turton. An interview with Lizard Squad, the hackers who took down Xbox Live. Dec. 2016. http://www.dailydot.com/debug/lizard-squad-hackers/.

[44] R. Vamosi. Study: DDoS attacks threaten ISP infrastructure. http://www.cnet.com/news/study-ddos-attacks-threaten-isp-infrastructure/.

[45] M. Weiser. Program slicing. In *Proc. ICSE*, Mar. 1981.

[46] J. Yan, S. Early, and R. Anderson. The XenoService – a distributed defeat for distributed denial of service. In *Proc. ISW*, 2000.

[47] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, 2003.