DeDoS: Defusing DoS with Dispersion Oriented Software

Henri Maxime Demoulin*
University of Pennsylvania

Tavish Vaidya*
Georgetown University

Isaac Pedisich
Bob DiMaiolo
University of Pennsylvania

Jingyu Qian
Georgetown University

Chirag Shah
University of Pennsylvania

Yuankai Zhang
Georgetown University

Ang Chen
Rice University

Andreas Haeberlen
Boon Thau Loo
Linh Thi Xuan Phan
University of Pennsylvania

Micah Sherr
Clay Shields
Wenchao Zhou
Georgetown University

ABSTRACT

This paper presents DeDoS, a novel platform for mitigating asymmetric DoS attacks. These attacks are particularly challenging since even attackers with limited resources can exhaust the resources of well-provisioned servers. DeDoS offers a framework to deploy code in a highly modular fashion. If part of the application stack is experiencing a DoS attack, DeDoS can massively replicate *only* the affected component, 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. Our evaluation results show that DeDoS incurs reasonable overheads in normal operations, and that it significantly outperforms standard replication techniques when defending against a range of asymmetric attacks.

CCS CONCEPTS

• **Security and privacy → Denial-of-service attacks**;

KEYWORDS

Denial-of-Service; Distributed Systems;

ACM Reference Format:
Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Bob DiMaiolo, Jingyu Qian, Chirag Shah, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. 2018. DeDoS: Defusing DoS with Dispersion Oriented Software. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3274694.3274727

*First Co-authors.

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274727

1 INTRODUCTION

Denial-of-service (DoS) attacks have evolved from simple flooding to pernicious asymmetric attacks that intensify the attacker's strength by exploiting asymmetries in protocols [33, 35, 36]. Unlike traditional flooding attacks, adversaries that perform asymmetric DoS are typically small in scale compared to the target victims. These attacks are increasingly problematic; the SANS Institute described "targeted, application-specific attacks" [33] as the most damaging form of DoS attack, with an average of four attacks per year, per survey respondent. Such attacks typically involve clients launching attacks that consume the computational resources or memory on servers. Types of asymmetric DoS vary, and are often targeted at a specific protocol.

An invariant of these attacks is that they exploit a fixed resource. For example, the SlowLoris/SlowPOST attacks function by establishing HTTP connections with the victim webserver, sending requests at a very slow rate to inflate their lifetime, consuming connection resources (e.g., file descriptors) at the target [37]. Similarly, the ReDoS attack uses specially crafted regular expressions that are slow to parse, amplifying the cost of serving malicious clients' requests [3]. Likewise, TLS Renegotiation attacks exploit an asymmetry in the SSL/TLS protocol: the server's cost of engaging in a handshake is about ten times that of a client [1].

While traditional *volumetric* attacks can be defended against by blocking transmissions on compromised machines [21], filtering traffic at routers [30], or detecting bogus requests at end hosts [23, 28], such approaches are ineffective against asymmetric DoS. Since asymmetric attacks tend to be relatively low-volume and often do not appear different from legitimate traffic, they can easily circumvent these defenses over time.

A straightforward defense mechanism against *asymmetric* attacks is simply to deploy more resources. This is often the de facto defense deployed in production systems: during an attack, the service is automatically replicated as virtual machines (VMs) or lightweight containers, on multiple machines to scale "elastically" to the extra load. Replicating *all* of the VM's or container's resources, regardless of which are being consumed, is enormously costly, making this approach unusable for most service providers. For example, if only a TCP state table is being exhausted (e.g., due to a SYN flood), the replication of an entire VM mitigates the attack, but does so at

an enormous overhead (since the TCP state table is a minuscule portion of the system's overall footprint).

In light of the limitations of existing defenses, we present a radically different approach called DeDoS that *defuses* DoS attacks via fine-granularity replication. We advocate software development in a modular fashion, such that components can be moved and replicated independently.

DeDoS provides a framework which allows programmers to construct more resilient applications through the use of fine-grained, modular components. Ideally, each component handles some small, focused aspect of an application that may be vulnerable to resource exhaustion. Example components include code for performing TLS handshakes or HTTP requests parsing. Crucially, with DeDoS, programmers do not have to worry about most of the deployment specifics: DeDoS offers an adaptive controller that makes real-time decisions on placing these components within physical resources (e.g., machines in a datacenter), and then adaptively clones, merges, or migrates them in order to meet service-level agreement (SLA) objectives. When SLA objectives are violated, this is treated as a potential attack, and individual components that are overloaded are replicated.

The DeDoS 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, DeDoS can respond to a TLS renegotiation attack by temporarily enlisting other machines with only spare CPU cycles to help with TLS handshakes. Second, the replication approach is not attack-specific and can thus potentially mitigate unknown asymmetric attacks. Once DeDoS 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 potentially allows a flexible and automatic response against even mixed attacks [25].

Specifically, we make the following contributions:

Architecture and design. We present the DeDoS architecture, outlining design challenges and our approach to create software as a dataflow of *minimum splittable units* (MSUs). We describe the API, communications, and synchronization components of DeDoS.

Our focus is on supporting new applications written in DeDoS' API: as services become increasingly modularized, they can either adopt the DeDoS API, or offload some critical functionality to DeDoS to mitigate asymmetric DoS attacks. For the sake of completing our argument, we also demonstrate how existing applications written in traditional or domain specific languages [26, 29] can be entirely ported to DeDoS.

Dynamic adaptation. We present strategies for assigning MSUs to physical machines, scheduling MSU executions assigned to threads, and using a global controller to make decisions on cloning and removing MSUs in the event of attacks.

Prototype implementation, case studies, and evaluation. As motivating use cases, we deploy three applications using our prototype implementation of DeDoS. These include a web server that we develop from scratch using DeDoS' dataflow API, and two existing software systems: a user-level transport library written in C that we port over to DeDoS, and routing software written using a declarative domain-specific language [29] that we compile into a DeDoS dataflow. Our evaluation results show that the overhead of DeDoS is comparable to equivalent code executed outside of DeDoS' runtime. Moreover, DeDoS is able to defend against a wide range of asymmetric attacks, maintaining significantly higher throughput for a much longer amount of time in the presence of changing attacks, comparable to traditional replication strategies.

DeDoS is not intended to be a cure against all possible DoS attacks. If an attacker can saturate a system's network links or completely consume the defender's resources, then the attack will still succeed. The goal of DeDoS is to better manage available resources to mitigate the attack. When it cannot completely defend against an attack, it aims to delay the attack's effects for as long as possible – ideally to the point where a human operator can put in place a longer term fix. DeDoS is also not a replacement or competitor of specialized defenses, such as hardware SSL accelerators [14]. These hardware-based approaches are more efficient than DeDoS because they are tailored to a particular attack vector, but are less generic in dealing with future unknown attacks (or combinations of attacks).

2 MOTIVATING EXAMPLE

We consider a 2-tiered web service hosted in a data center, where an HTTP server queries a database server in response to users' requests. The attacker launches a TLS renegotiation attack [1] that consumes CPU cycles on the HTTP server. Hence, legitimate requests are being served very slowly, or not at all. In this typical asymmetric attack, the attacker is unable to overwhelm the defender's network bandwidth, but succeeds by exhausting other resources (here, CPU cycles).

Our goal is to *automatically mitigate such an attack, even if it has a new attack vector*, and to maintain quality of service (QoS) to the legitimate clients. DeDoS is not specifically designed to defend against brute-force volumetric attacks that saturate a data center's ingress link, or exploits that take over data center machines.

2.1 Strawman solutions

One possible defense against DoS attacks is to *filter* or *block* suspicious network traffic – either based on source addresses, specific traffic content or other traffic characteristics. However, this relies heavily on request classification, thus is susceptible to false positives and negatives. Moreover, it is difficult to differentiate between legitimate spikes in traffic and actual attacks.

Another approach is to increase resource capacity via *replication*. For instance, to handle a TLS renegotiation attack, an operator can launch more web server VMs to sustain more connections. This defense does not depend on accurate attack detection, but it can be inefficient. In the TLS renegotiation example, even though the attack is limited to the key generation logic (and thus stressing CPU usage on the host), naïve replication replicates the *entire* web server, unnecessarily wasting non-affected resources such as memory.

2.2 DeDoS solution

We observe that overall, data centers machines are under utilized [11], but current software architectures cannot effectively use them. In our example, the database servers' CPUs will be mostly idle while

the web servers' CPUs are overwhelmed. If the former's CPUs were able to alleviate the load on the latter's by contributing their computational power, the capacity at the bottleneck (TLS handshake) would increase.

Achieving this requires designing application stacks as smaller functional pieces that can be replicated and migrated independently. This additional flexibility would enable an attacked service to use *all* of the available datacenter resources for its defense by temporarily enlisting other machines running different services, resulting in a substantial increase in the service's capacity and achieving better QoS for legitimate clients.

For example, in TLS renegotiation, instead of replicating the entire web server, we can instead replicate only the key generation logic. If the database servers have spare CPU cycles, they will be able to accommodate execution of this logic and alleviate the CPU bottleneck caused by the attack. In contrast, naïve replication would not work when the database servers lack the entire set of resource required to run additional HTTP servers. In other forms of asymmetric attacks that exhaust other types of resources (e.g. memory), one can adopt the same approach, in this case, replicating the memory intensive component into other machines that have spare memory.

3 DEDOS DESIGN

A DeDoS application consists of several components called *minimum splittable units (MSUs)* (Figure 1). Each MSU is responsible for some particular functionality. For instance, a web server might contain an HTTP MSU, a TLS MSU, a page cache MSU, etc. (Figure 1a).

Related MSUs communicate with each other. For instance, HTTPS requests may enter the system at a network MSU, be decrypted by the TLS MSU, and parsed by the HTTP MSU. Collectively, the MSUs form a *dataflow graph* that contains a vertex for each MSU and an edge for each communication channel (Figure 1b).

Each DeDoS deployment contains a central *controller* which provides an API for programmers to deploy their application. The controller can either receive a pre-computed allocation, or perform an initial allocation plan to decide how many instances of each MSU should exist, and which machines they should run on, based on the requested performance requirements and available resources (Figure 1c).

Additionally, the controller continuously collects runtime statistics about available resources and the performance of each MSU. If it detects that some MSU instances are overloaded (e.g., due to an unknown attack; Figure 1d), it can create additional instances of these MSUs, placing them on machines where resources are still available (Figure 1e). Thus, the data center can defend itself against the attack with all available resources, not merely the ones that happen to be "in the right place."

3.1 Minimum splittable units

When designing an application for DeDoS, the question of defining the granularity and boundaries of MSUs arises. While smaller MSUs can result in a more precise response during an attack—since it allows DeDoS to replicate only the functions that the adversary is actually targeting—*too* small and numerous MSUs can result in unacceptable overheads because of the delay introduced on the execution path. This tradeoff has already been unveiled in the past with, for example, the fall of mainframe computers and the rise of microservices [16, 19].

The general approach we advocate is based on the microservices design [31]: MSU split points are appropriate when there are loose couplings between components, functional domains are clearly encapsulated, and individual components are provably stables.

For known attacks, it is also advantageous to purposefully demarcate MSUs to most optimally respond to the potential attack. For example, to protect against a SYN flood, the portion of the TCP stack that handles TCP connection state could be isolated into its own MSU.

However, a key benefit of DeDoS is that it does not require apriori knowledge of the attacks it defends against. Hence, in many instances, programs may not be perfectly spliced to optimally match a novel attack. Indeed, this is our expectation and observation in practice. In such instances, MSUs may contain features unrelated to the attack, resulting in non-optimal resource allocation. However, we emphasize that such duplication will *always* be preferable and is very likely far better than naïve replication. In general, we posit that splitting software components following a microservices-like programming paradigm will yield significant protection against DoS while incurring limited overheads. We empirically measure these overheads in a number of applications, constructed using this design pattern, in §7.

3.2 Inter-MSU communication

MSUs communicate with each other by exposing an API that can be called by other MSUs. The API functions are asynchronous and one-way. This enables efficient event-driven implementations (analogous to SEDA [39]). If a call needs to return a value, this is handled by another call in the reverse direction.

Communicating MSU instances can reside on different machines. DeDoS makes this transparent to the MSUs by injecting a bit of "glue code" that converts calls into a local function call (if the callee is on the same machine) or a network packet (if the callee is remote).

3.3 Routing tables

When an MSU instance of type X wants to invoke a function on another MSU of type Y, X does not need to know where the instances of Y are currently located. DeDoS handles routing by maintaining a *routing table*, configured by the controller, which contains information about MSU types and implements customizable load balancing policies and routing functions. By default, DeDoS spreads the load evenly among MSU instances of the same type, enforcing instance affinity to related packets (e.g., from the same flow or user session). As we show below, we can extend this policy to implement queue-length based routing.

3.4 DeDoS runtime API

So far, we have treated the dataflow graph as largely static. However, the controller can also dynamically create new MSU instances. To make this possible, each machine in a DeDoS deployment runs the DeDoS *runtime*. When it is first started, the runtime process is an empty shell: it contains the code for all the MSUs that the system could create, but none of this code is active yet. The runtime listens

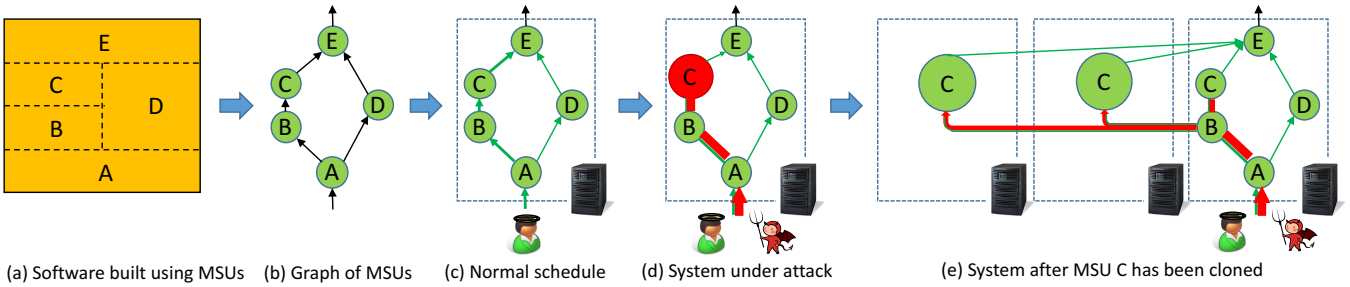

Figure 1: Example use case of DeDoS. The software is built using MSUs (a), represented as a dataflow graph (b). MSUs are then scheduled on the available machines (c). When an attacker attempts to overload one of the components (d), DeDoS disperses the attack by generating additional instances on other machines (e).

for commands from the controller. The `add` command creates a new MSU instance, while `remove` deletes one. Those operations involve adjusting the routing table of connected MSUs. MSU also expose an API, to `execute` their main function, or access their internal state. The full API is documented online [8].

3.5 Support for existing applications

DeDoS is a new platform which aims at improving applications' resilience from the very beginning of their development process. Consequently, our focus is on enabling *new* applications using DeDoS' model. Nevertheless, we do not require applications to be written from scratch in order to benefit from DeDoS' defense mechanisms. We provide a proof of concept in our case study (§6) by splitting a user-level TCP stack into MSUs. Since DeDoS does not require the entire software to be partitioned, rewriting existing code can start small by only carving out the most vulnerable component while the rest of the application runs as a single MSU.

We note that it is possible to – partially or fully – automate the partitioning. Some domain-specific languages are already written in a structured manner that lends itself naturally to this approach. For instance, a declarative networking [29] application can be compiled to an MSU graph that consists of database relational operators and operators for data transfer across machines (see §6). Work in the OS community [19] has shown that even very complex software, such as the Linux kernel, can be split in a semi-automated fashion.

4 RESOURCE ALLOCATION

To ensure that the applications meet their SLAs, DeDoS needs a way to make and enforce resource allocations to MSU instances at runtime. DeDoS performs resource allocation at two layers: each machine schedules MSU instances locally based on their resource needs, whereas a central *controller* is responsible for decisions requiring a global view, such as cloning or merging MSU instances. To enable runtime adaption, each machine has an *agent* that continuously monitors local MSUs, periodically submits statistics to the controller, and is responsible for handling the controller's commands.

4.1 Machine-local scheduling

When an application is divided into a large number of fine-grained MSUs, switching from one MSU instance to another is a very frequent operation, and we cannot afford to enter the kernel every time. Because of this, we privilege user level scheduling and context switching, using a set of kernel threads that are each pinned to a particular core. This approach has the additional advantage that it does not require changes to the kernel.

DeDoS schedules MSUs at the granularity of events. On each core, DeDoS maintains a local scheduler, and a "data queue" for each of the local MSU instances, which stores the incoming messages. Whenever the core is idle, the scheduler thread picks an MSU instance according to a chosen policy, picks a message from that MSU instance's data queue, delivers that message, and waits for the MSU instance to finish processing it. Scheduling is partitioned and non-preemptive—cores do not "steal" messages from other cores and they do not interrupt MSU instances while they are processing messages. Partitioned scheduling avoids inter-core coordination in the general case and thus keeps context-switching fast.

By default, DeDoS uses a round-robin policy, which picks at most r_i messages from data queue i and then moves on to data queue $i + 1$. The parameters r_i can be adjusted by the controller at runtime, e.g., based on the relative load of the MSU instances. We design worker threads such that each can implement specialized policies (e.g., Earliest Deadline First – EDF).

4.2 Initial MSU assignment

When a DeDoS deployment is first started or a new application is launched, the controller finds an initial assignment of MSU instances to machines, such that the application's SLA goals (throughput and end-to-end latency) are met. The assignment must be *feasible*: each machine must have sufficient resources (e.g memory) to execute the MSU instances that are assigned to it. To find such an assignment, we can formulate them as constraints and use an existing solver or bin-packing heuristic.

Since the controller cannot predict the effects of future attacks, the computed assignment only maintains the SLAs in the absence of an attack. However, the controller monitors the system at runtime, using the statistics that are submitted by the agents. If it detects that the SLAs are being violated (e.g., due to a DoS attack), it adjusts the assignment to mitigate the effects of an attack, using the process we describe next.

4.3 Cloning and merging

The controller supports customizable cloning and merging policies. The DeDoS native heuristics (i.e., default policies) operate by analyzing the collected metrics; forming a decision about whether to clone, merge, or do nothing; and executing the chosen action.

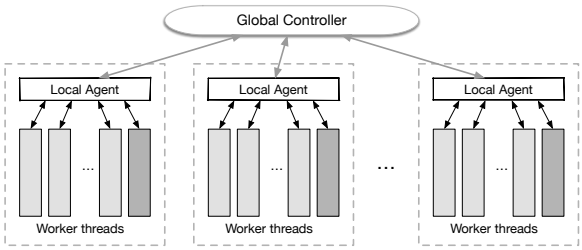

Figure 2: DeDoS architecture. A controller manages local runtimes (represented by dotted boxes).

As native policies, the controller attempts to clone an MSU if its minimum reported queue length is greater than 0. The rationale here is that the system is provisioned such that the expected arrival rate is sustained, and in such conditions, no queue should be built for any of the MSU instances. In addition, if all runtimes are utilizing more than a configurable percentage of a resource (e.g., memory) and an MSU type accounts for a configurable percentage of a runtime's utilization of that resource, the controller will begin to clone MSUs of that type. This latter policy targets the system's bottlenecks by increasing parallelization.

Once the decision to clone is made, the controller picks a satisfying machine for the new instance, favoring locality with the clone's neighbors in the dataflow graph. A local machine is best to minimize network communication. Once a machine has been elected, the controller picks the least loaded core which does not already host an instance of the same type, and contacts the machine's agent to spawn the instance. The controller also updates all the relevant routing tables to enforce the load balancing policy in place for this MSU type.

An attempt to clone will fail if it is not possible to place the clone on any available runtime or if the same type of MSU has been recently cloned.

The controller removes cloned MSU instances when they are no longer needed (e.g., when an attack ends). Two conditions must be met for an MSU to be removed. First, the last runtime where a clone has been placed must report a maximum queue length of 0 in the last monitoring interval (following the rationale described earlier); second, the MSU type must not be significantly contributing to more than a configurable percentage of a resource consumption on *any* runtime. An attempt to remove will fail if an attempt to clone an MSU of that type was made in the recent past (to protect against system oscillation), or if some configurable amount of time has passed since the last removal of that type.

5 IMPLEMENTATION

The DeDoS implementation consists of 10K lines of C code, and is available on GitHub [8] under GPLv3.

5.1 Overview

Our prototype (Figure 2) consists of two key components: a *controller* that orchestrates all other machines and a *local runtime* running on each machine.

Controller: The controller performs load balancing and responds to DoS attacks. It takes as input DeDoS applications in the form of a graph and assigns MSUs to machines based on the placement algorithm described in §4. MSU cloning is on-demand and automated.

We set the following default parameters for DeDoS management policies (see §4.3): the controller clones an MSU type if all runtimes are utilizing more than 40% of the memory or file descriptor (FDs) pool, and the type accounts for at least 50% of its runtime utilization of that resource; for removal, the MSU type must not be contributing more than 40% of the memory or FDs pool on any runtime. Removal fails if an attempt to clone an MSU of that type was made in the last 20 seconds, or if less than 5 seconds elapsed since the last removal of that type or its dependencies. Those parameters are based on our domain expertise of how our testbed performs.

The controller also establishes routing policies. Within a route toward an MSU type, endpoints are weighted proportionally to the ratio of all requests enqueued for that type, over the endpoint's queue length (effectively the inverse of the occupancy ratio of the endpoint). This allows DeDoS to load balance requests across all instances of a type.

Local runtime: Each local runtime schedules and executes MSUs. With some notable exceptions explained below, the local runtime maintains a POSIX thread for each CPU core, which schedules the MSUs for execution. The specific MSU-to-thread bindings are determined by the controller.

Controller-runtime communication: Each runtime maintains long-lived TCP connections to the controller and every other runtime instance. These connections are carried over an isolated management network, and are used to manage MSUs and to pass data between remote MSUs. Additionally, the local agent at each runtime periodically gathers MSU and system statistics (such as queue length, execution time, number of file descriptors in use, etc) that are then sent to the controller.

5.2 DeDoS local runtime

The internal design of the DeDoS runtime consists of MSUs, worker threads, and local agents.

MSUs: MSUs are constructed as a collection of C/C++ functions with a well-defined API. Each MSU maintains a *data queue* that contains incoming requests. MSUs are executed within worker threads (explained next) that persist in the local runtime. As output, an MSU may enqueue messages onto the data queues of other MSUs. This is handled by efficient pointer manipulation when source and destination are co-located, or by long-lived TCP connections otherwise.

Worker threads: DeDoS proposes to either pin POSIX threads to CPUs or not. Pinned threads are used for operations that avoid blocking system calls, such as reading from non-blocking sockets or TCP state table manipulation. A pinned worker thread may be assigned multiple MSUs (its *MSU pool*). Pinned workers do not involve the kernel's scheduler and allow DeDoS' operator to implement their own scheduling MSU-aware algorithm. Pinning also maximizes CPU utilization and reduces cache misses that would otherwise occur if MSUs were migrated between cores.

In more detail, pinned threads run a scheduler (§4.1) that continuously (1) picks an MSU from its pool, (2) executes it by dequeuing one or more item(s) from its data queue and invoking the `execute`

function of the MSU's API, and (3) repeats. Pinned threads currently schedule MSUs in a round-robin fashion.

In our current implementation, we assign MSUs that have blocking operations (e.g., disk I/O) to their own non-pinned threads, such that they are scheduled by the Linux kernel as generic kernel-level threads. In later versions of DeDoS, we anticipate supporting MSU preemption and resumption, which will allow blocking MSUs in the MSU pools of pinned threads.

Each worker thread keeps statistics on resource usage of each of its MSUs, and global metrics such as their data queue lengths and number of page faults. In addition, MSU's API allow programmers to implement custom metrics (e.g., frequency of access of a given URL in an HTTP MSU). Those statistic are then gathered by the local agent (explained below) to be sent to the controller.

Finally, each worker thread periodically runs an *update manager* that processes the thread's *thread queue*. Unlike the MSU data queues, the thread queue is solely for control messages, and is itself populated by the local agent. It stores requested configuration changes such as creating and destroying MSUs. In effect, the thread queue serves as a buffer of requested changes, and avoids the overhead of locks and other consistency mechanisms that would otherwise be required if the local agent directly manipulated the worker threads' data structures.

Local agents: Each runtime operates a *local agent* in its main thread. The local agent is responsible for communicating with the controller and other DeDoS runtimes over long-lived TCP connections. In particular, the local agent receives commands for configuration changes from the controller and routes them to the appropriate worker threads. In addition, at a configurable interval which we set to 100ms, the local agent gathers statistics from all the threads and forwards them to the controller.

6 CASE STUDIES

We demonstrate the feasibility and applicability of DeDoS by considering three case studies: a web server written using DeDoS' MSU interface; an existing userspace protocol stack ported to DeDoS; and an application written using a declarative domain specific language [29] that has been translated into a DeDoS dataflow.

Web server. For our first case study, we implemented a simple web server constructed as five MSUs. The *I/O MSU* accepts incoming requests and steers them toward the *Read MSU*, which performs the TLS handshake, deciphers data, and relays plaintext to the *HTTP MSU*. The HTTP MSU implements NodeJS's HTTP Parser [24]. Once the request is parsed, the HTTP MSU issues a call to the database tier to retrieve some object file, then enqueues the request to a *Regex Parsing MSU*, which uses the PCRE engine to parse it. The final HTTP response is sent to the *Write MSU*, which wraps it in a layer of TLS and sends it back to the client. We use OpenSSL version 1.0.1f for TLS support in both Read and Write MSUs.

Importantly, with the exception of the I/O MSU, all of the web server's MSUs are event-driven and non-blocking. We favor non-blocking MSUs to augment the overall utilization of our machines. To avoid having to migrate socket states between machines, we configured the controller to enforce that the I/O, Read, and Write MSUs reside within the same DeDoS instance for a given client connection. We use HAProxy [38] as a front-end load balancer, allowing us to direct incoming client connections to any I/O MSU on any DeDoS instance.

Our web server leverages DeDoS' fine-grained modular architecture to mitigate DoS attacks. In §7, we demonstrate the resilience of our application against three DoS attacks: TLS renegotiation attacks [1], ReDOS attacks [3], and HTTP SlowLoris attacks [37].

Userspace network stack. Our second case study consists of an existing software project that we ported to run on DeDoS with minimal effort. PicoTCP [34] is an open-source userspace TCP stack written in approximately 33,000 lines of C code (as reported by SLOCCOUNT). We chose PicoTCP since it is well-structured and written in a modular fashion, making it easy to manually determine cut-points (i.e., to form MSUs).

We have separated out a standalone handshaking MSU from PicoTCP. When a SYN flood attack occurs, the TCP Handshake MSU is replicated into multiple copies on the same or different machines. Load-balancing across these clones is achieved by using a consistent hashing scheme within the PicoTCP MSU: based on a hash over the incoming packet's four-tuple (source and destination addresses and ports), DeDoS performs load-balancing by distributing the handshaking requests (in the form of SYN, SYN/ACK, ACK packets) to the various TCP Handshake MSU instances. Packets belonging to the same three-way handshake (e.g., the client's SYN and ACK) are routed towards the same TCP Handshake MSU, obviating the need to transfer state.

Given the modular nature of the PicoTCP code, separating the TCP stack into separate MSUs was fairly straightforward. The bulk of our efforts lay in wrapping PicoTCP's "main loop" within an MSU to allow DeDoS's runtime's scheduler to execute the MSU according to its scheduling policy. Within the PicoTCP MSU, we added functionality to re-inject SYN-ACKs generated by Handshake MSUs into the PicoTCP stack for them to be sent back to the client, and code to restore TCP state received from Handshake MSUs (for successful connections) into PicoTCP's internal TCP state data structure.

In summary, with only minor modifications, we transformed the monolithic PicoTCP application into a DeDoS-enabled version in which handshake components could be replicated on demand, both within the local machine and on remote DeDoS instances. Overall, we changed less than 0.1% of PicoTCP's original codebase.

Declarative packet processing. We also consider an application that is written in a domain specific language. We select an application that does routing (packet forwarding), written entirely as a declarative networking [29] program. Declarative networking programs are written in a variant of Datalog called *Network Datalog* (or NDlog). An NDlog program consists of a set of rules, where each rule is of the form `h :- b1,b2,. . . , bn`, indicating that a *head tuple* is generated so long as all *body tuples* b1, b2, . . . , bn are available. For example, the rule `packet(@Y,A,Data) :- packet(@X,A,Data), Neighbor(@X,A,Y)` results in all packets arriving at X being forwarded to neighbor Y based on some attribute A (e.g., the packet's header data). Declarative networking has been adopted for network forensics, datacenter programming, and overlay routing.

Since these programs have their roots in the database relational model, they can be compiled into an MSU dataflow of relational

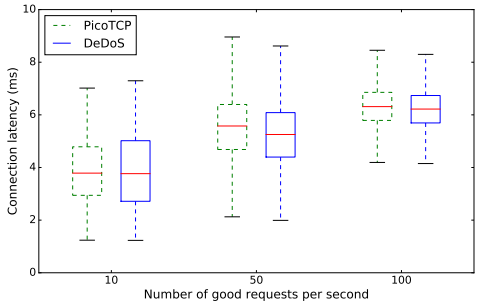

Figure 3: Connection latency for standalone PicoTCP and DeDoS-enabled PicoTCP ("DeDoS").

operators. For example, the body tuples are executed as a series of pair-wise database *join* operations, additional filters in the form of *selection* operators, and the head tuple is generated as a *projection* operator. The generated tuple may be sent to the same or different machine using DeDoS. We find such automatic translation to be a promising method of adopting existing applications to DeDoS.

7 EVALUATION

In this section, we aim to answer two high-level questions through several sets of experiments: (1) does DeDoS run with reasonable overheads in normal operation, and (2) how well can DeDoS defend against asymmetric DoS attacks, as compared to whole-system replication? Our experimental testbed consists of a cluster of 8 machines connected via a 10 Gbps switch in a star topology. Each machine has 8 1.80 GHz cores (with hyperthreading and DVFS disabled), 64 GB of memory, and runs Linux kernel 4.4.0-62. We note that a video demonstration of DeDoS is available online [2].

7.1 Overheads

To measure the overhead introduced by the DeDoS runtime during normal operation, we run two applications described in §6 within and outside of DeDoS on a single server machine.

Web server: We compare a DeDoS web server to a standalone web server with the same implementation but compiled as a monolithic application outside of DeDoS. Our workload consists of HTTPS requests generated by Tsung [4] at an exponentially distributed rate for a period of five minutes, with a mean of 2500 requests per second (r/s). We average latencies over intervals of one second. The standalone webserver has mean latency of 43ms, and 1.8ms standard deviation. DeDoS' webserver has a mean latency of 48.5ms and 12.3 standard deviation. This accounts for a mean 10.5% overhead introduced by DeDoS, which is caused primarily by the enqueuing and dequeuing of data across MSUs. Because in their current implementation, worker threads on DeDoS do not "steal" work from remote queues, there are some rare instances of MSUs sitting idles while others are building a queue, hence the higher number of outliers with DeDoS.

PicoTCP: We compare a DeDoS-enabled version of PicoTCP to the standalone monolithic ("vanilla") version. The DeDoS-enabled version uses a single worker thread on a single runtime and has two MSUs: a handshake MSU and the remainder of the PicoTCP stack as a separate MSU. As an application, we use a simple echo server that mirrors back incoming requests.

We consider *connection latency*, the time required to complete a TCP handshake as measured by the client. We measure the connection latency over a 15-minute period. During this time, the client continuously creates new TCP connections at a steady rate, sends (and receives) 32 bytes of data, and disconnects. Figure 3 shows the distribution of connection latencies for the vanilla and DeDoS-enabled versions of PicoTCP under different client request rates. The DeDoS-enabled version incurs a modest 5.5% increase in connection latency.

Next, we measure the throughput that both TCP stacks can achieve. We create a number of different clients that simultaneously access the echo server. Each client repeatedly sends and receives 1024 bytes, with a 10ms pause between transmissions. We observed that there is no significant difference between the throughput that both stacks can achieve. PicoTCP and DeDoS both reached their maximum bandwidth of 57.66 Mb/s and 57.71 Mb/s respectively around 100 simultaneous connections and the throughput remained similar for more than 100 simultaneous connections. (The absolute numbers are low because PicoTCP is a userspace network stack, designed for portability instead of maximum performance. Our goal here is to measure the overhead of the DeDoS runtime.)

Overall, our results suggest that running applications within the DeDoS runtime does not significantly change the throughput or the latency it can achieve.

7.2 Attack mitigation

To evaluate the efficacy of DeDoS in mitigating DoS attacks, we launch ReDOS, TLS renegotiation, SlowLoris, SYN flood, and a volumetric flood attack against our applications. The first three are launched on the webserver, while the latter two are on PicoTCP and an NDlog program respectively.

7.2.1 Attacks against webserver . We configure our testbed for attacks against our webserver as follows: the webserver is deployed on three machines while three other machines run instances of an in-memory database. All web requests access the database, and HAProxy is used to distribute HTTP and database requests. The remaining two machines are used to generate legitimate ("good") and attack traffic respectively. To demonstrate DeDoS' ability to defend against changing attacks and reclaim resources, under dynamic traffic patterns, we run a two hours long experiment during which attack durations are *randomly* distributed. Good traffic is generated by Tsung, and exponentially distributed. We simulate diurnal variations by setting Tsung's distribution mean at 1500 r/s, and increasing by slices of 500 r/s up to 3000 r/s, at which point it gradually decreases back down to 1500 r/s. Tsung's requests are configured to time out after 1s if they cannot connect. When no attack occurs, clients experience average latencies of 50ms. For attack traffic, we develop a C client which generates malicious ReDOS and TLS renegotiation requests, and use an existing Python-based SlowLoris [20] attack tool.

Figure 4 shows our main findings for different attacks on the HTTP servers for a single run of the experiment than ran continuously for 2 hours. The top figure shows response times for successful connections averaged every second, while the middle figure presents the connection success rate during this experiment. The bottom figure shows the number of MSU instances of a given type

deployed on the system over time. Attacks occur during the period colored in red. We compare DeDoS to two other approaches: (1) an approach that does not replicate at all under attack ("standalone"), and (2) an approach that naïvely replicates an entire webserver to one of the database servers when under attack ("naïve"). Initially, the DeDoS' webserver has 4 Read and 2 Regex MSUs on each of the three starting machines.

During the entire course of the experiment, DeDoS is autopiloting without inputs from human users. We observe that DeDoS can accurately detect and react to the injected attacks based on the resource allocation polices described in §4 without apriori knowledge of the attacks. DeDoS can consistently *and automatically* decide on an effective mitigation strategy against different types of attacks. Figure 4 shows that DeDoS consistently outperforms standalone and naïve approaches, and sustains low latency and high response rate while standalone and naïve can only provide limited or sporadic services.

TLS renegotiation attack: This attack consumes the victim's CPU by having malicious connections repetitively triggering TLS handshakes. In our setup, a single handshake requires about 2.1ms computation time (we use a 2048-bits RSA key), and every malicious request triggers 100 renegotiations before closing. During the first TLS renegotiation attack in Figure 4, the attacker increases the strength of the attack from 1 to 100 r/s over a period of 13 mins. At the start of the attack, standalone performs better than DeDoS until CPUs get overwhelmed by attack requests (around 75 r/s); it increases the average latency for good requests to the order of seconds. Naïve replication performs even worse and causes connection success rate to drop to almost 0% once the entire webserver has been replicated to the database machines. This is due to paging that occurs on the database server as a result of the additional memory footprint imposed by the cloned webserver. Even successful connections experience latency on the order of tens of seconds.

During the attack, DeDoS' controller observes abnormal levels of pending requests in the system, and gradually increases the number of Read MSUs from 12 to 39 (1 more on each original machine, plus 8 per database machine). Unlike naïve replication, Read MSUs have a low memory footprint and do not cause paging on the database machines. This results in average latency of 70ms for good requests during attack.

Once the attack stops, the DeDoS controller observes that the conditions explained in Section 4.3 are met, and reclaims resources by tearing down the cloned MSUs.

The second TLS attack in Figure 4 shows the performance of DeDoS under a steady state attack with 100 r/s instead of a gradual increase in attack strength. Under this relatively hight attack strength, CPU resources for standalone are quickly overwhelmed, and connection success rate for good clients falls to 50% with 3s latency on average. DeDoS applies the same policies for resource management, maintaining 39 Read MSUs, and while its performance drops momentarily, it manages to serve good clients with an average latency of 70ms.

ReDOS attack: In this attack, each malicious request issues a complex regular expression operation that exploits a PCRE vulnerability [7], requiring approximately 100ms of computation time. The first ReDOS attack increases attack strength from 1 r/s to 200 r/s and lasts 9 mins. Similar to TLS renegotiation, standalone initially does better than DeDoS until CPUs are overwhelmed by malicious requests. On the other hand, DeDoS gradually increases the number of Regex MSUs (up to 27 new instances) and maintains 100% success rate, but with an increased average latency of 150ms. We observe much less variations in the number of Regex MSU than Read MSU because of the nature of the workload: TLS handshakes are much shorter, and performed over non-blocking I/O, while the regex parsing operating cannot be preempted by DeDoS. The second ReDOS attack is performed at a steady rate of 200 r/s over 11mins. Standalone clients almost instantly experience average latencies on the order of seconds after the attack is launched. DeDoS, while initially overwhelmed as well, quickly recovers by spawning 27 new Regex MSUs, managing to keep the latency on the order of tens of milliseconds.

HTTP SlowLoris: This attack targets the connection pool of the webserver by exhausting the file descriptors (FDs) available for the process. The attack works by opening a connection to the server, and slowly sending HTTP headers one after the other, at such a pace that the server keeps each connection open for a significantly longer time than usual. We configure our kernels to allow each process to open 2^{13} concurrent FDs (from an initial value of 2^{10}). We configure the attack tool to open up to about 41K concurrent connections to the webserver during 17mins. Standalone is able to withstand the attack until the FDs limit is reached (in about 220 seconds). Then the connection success rate quickly drops to about 3 r/s, and the good requests experience a sharp increase in latency, since the webserver threads are kept busy with processing HTTP headers that are continuously sent from malicious clients. DeDoS, on the other hand, is able to spawn 22 new Read MSUs on each of the database server, increasing its global file descriptors pool, and allowing it to sustain 100% successful connection rate. Due to paging, naïve is unable to respond to a majority of the connections.

7.2.2 Additional attacks. In addition to the attacks discussed on the web server, we discuss two more attacks and their mitigation using DeDoS.

SYN flood attack: Our SYN flood experiment consists of a number of "good" (i.e., non-attack) clients accessing an echo server built on top of PicoTCP. Each good client attempts 10 requests per second, where each request establishes a TCP connection, sends and receives 32 bytes of data, and then closes the connection. A TCP connection is considered successful only if the handshake completes within 60 seconds. The SYN flood is launched after one minute of normal traffic, runs for three minutes, and then stops. We use HPING3 to launch SYN flood attacks and vary the intensity of the flood. The experiment continues for an additional two minutes (during which no attack occurs) to observe the recovery period.

Since under normal conditions, an application's use of TCP is tightly coupled to (i.e., inseparable from) the machine's local network stack, standalone PicoTCP is assigned a single process on a single machine. We set the size of its connection buffer to 1MB, corresponding to 26,214 pending connections. We note that this limit is significantly larger than the 2^{10} pending connection limit offered by default on Linux.

In contrast, DeDoS can mitigate a SYN flood by cloning MSUs, potentially on other hosts. Our DeDoS-enabled version of PicoTCP consists of separate MSUs for performing the handshake and for

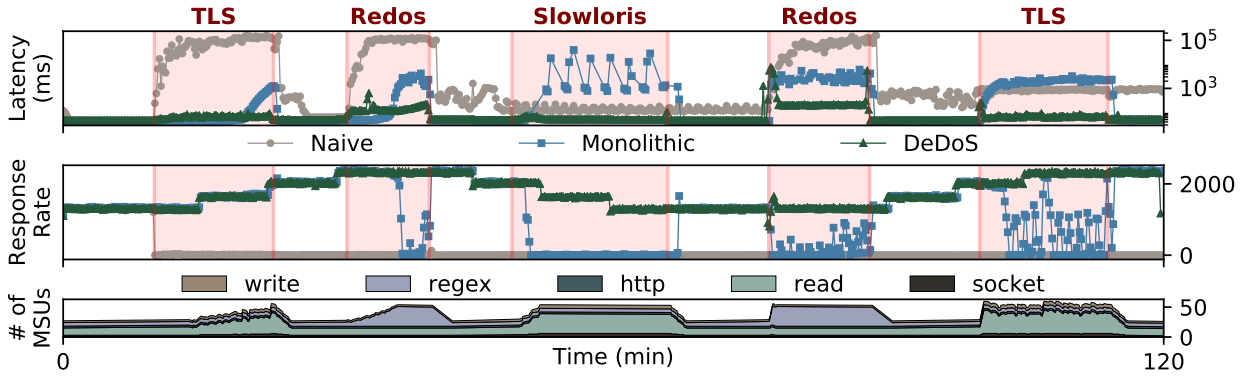

Figure 4: Requests latency and success rate during TLS renegotiation, ReDOS, and SlowLoris attacks on the web server.

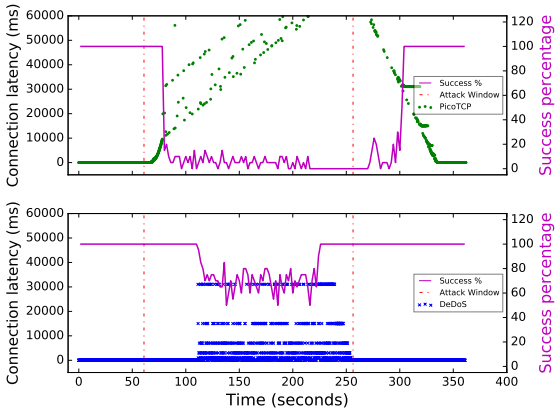

Figure 5: Handshake latency of good clients under a SYN flood (2000 SYNs/sec) with standalone PicoTCP *(top)* and DeDoS with 3 Handshake MSUs *(bottom)*. Vertical lines denote the start and end of the SYN flood. The right y-axis plots the good clients' average percentage of successful TCP handshakes.

transferring data. Each instance of a Handshake MSU is provided with a 1MB connection buffer. We operate the PicoTCP MSU (the non-handshake related portion of TCP) on a single machine that also hosts the echo server. We use three other physical machines in our cluster and spawn a maximum of three additional Handshake MSUs per machine. To evaluate the efficacy of DeDoS, we vary the number of Handshake MSUs (by manually overriding the controller's actions) and measure system performance.

We consider the success rate of TCP handshakes during the interval between the first and last instances in which a TCP connection failed, as observed by a good client. This reflects the steady state of the attack and avoids the "ramp up" period in which the attack has not yet become effective.

Our results show that DeDoS is able to provide superior service throughout the attack. Figure 5 shows the connection latency of good clients during a 2000 SYN/second attack. On the second y-axis, we show the average percentage of successful TCP handshakes ("success percentage") computed over a two-second interval. PicoTCP (top graph) fails to service good requests as soon as the attack starts—the percentage of successful TCP handshakes almost immediately drops to below 10%. The few connections that are successful experience very high latencies (first y-axis).

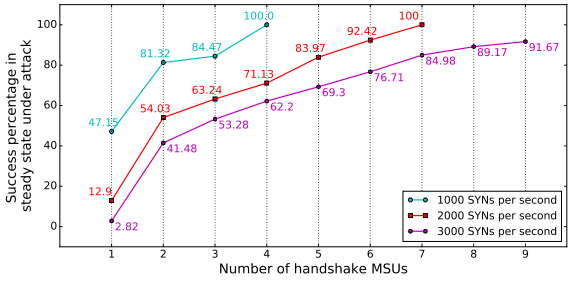

Figure 6: Percentage of successful TCP connections with varying numbers of Handshake MSUs during a SYN flood.

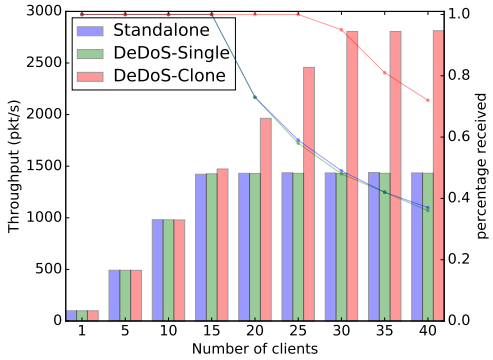

Figure 7: The throughput of the declarative packet processing application when under DoS attack.

In contrast, with three cloned Handshake MSUs running on the same physical host (bottom graph of Figure 5), DeDoS is able to achieve an average success percentage of approximately 64% during the steady state of the attack and recovers quickly after the attack ends. (The stratified "bars" in the figure are due to TCP retransmissions and TCP backoff.)

Figure 6 shows the scalability of DeDoS and the improved response to various SYN floods with increasing resources. For a given attack strength, DeDoS is able to serve more legitimate requests as the number of handshake MSUs increases. Here, Handshake MSUs are equally distributed across the cores on three physical machines. Notably, we are able to completely mitigate the attack (as measured

by successful client TCP connections) for moderate attack rates of 1000 and 2000 SYNs/second with four and seven MSUs, respectively.

Declarative packet processing attack: In our final experiment, we launched an attack against the declarative packet processing application described in §6. We initialized the packet processing application with a large in-memory neighbor table, rendering naïve replication too expensive in this case. Our workload consists of a varying number of clients that forward packets via our application. We increase the attack rate by using more clients to send more traffic. Figure 7 shows the throughput (pkts/s) that can be processed by (i) a standalone implementation, (ii) a DeDoS-enabled application with cloning disabled, and (iii) a normal DeDoS-enabled application. As before, the results show that standalone and DeDoS achieve comparable throughputs (which indicates low overhead), but that cloning enables DeDoS to handle roughly twice as many clients during an attack.

8 RELATED WORK

Volumetric attacks: Most existing DoS defenses focus on volumetric attacks, e.g., the attack on Dyn's DNS service [18]. Zargar et al. [41] provides a detailed survey. These defenses are orthogonal to DeDoS, whose main focus is *asymmetric* attacks. They are also complementary to DeDoS, and can be deployed together: for instance, we can deploy traditional traffic scrubbing as an initial defense to filter out certain suspicious traffic, and then use DeDoS to handle the attack traffic that cannot be easily recognized as suspicious; moreover, the fine-granularity cloning strategy in DeDoS can also help mitigate volumetric attacks as well.

Amplification attacks: Newer attacks, such as reflective denial-of-service (DRDoS) attacks exploit network protocols to launch amplification-based attacks [35]. DeDoS may be useful for defending against these attacks, and we plan to investigate this in future work.

Dispersion-based defenses: DeDoS [13] is a type of dispersion-based defense against DoS attacks. Load balancing strategies [27, 30, 32] can also disperse the effect of DoS attacks, but they tend to require a significant amount of redundancy and is costly in terms of resource management. DeDoS is also inspired by the SplitStack architecture [12]; relative to SplitStack, DeDoS comes with novel schemes for resource management and automated cloning, a concrete implementation, as well as a thorough experimental evaluation.

Cloning-based defenses: XenoService dynamically clones websites when they are under attack [40]. Bohatei [15] also dynamically launches more VMs to defend against known attacks. Similarly, Jia et al. [22] describe a technique that attempts to conceal the location of replicated services from an adversary. All these approaches use whole-system replication of services, which offers less protection than DeDoS because of the significant resource waste. Unlike existing approaches, DeDoS does not attempt to recognize legacy attacks and deploy pre-developed defenses; instead, DeDoS dynamically responds to *new* attacks by cloning just the system components that are under attack.

Function-as-a-Service and Micro-services platforms: DeDoS is conceptually related to the trend toward fine-grained granularity decomposition of functions seen in FaaS platforms [17]. However, those platforms [5, 6, 9, 10] have often constraints that make them unsuitable for the deployment of stateful, long-lived services. Similarly, DeDoS is *not* a Micro-services platform. We envision that DeDoS can be integrated to those platforms.

9 CONCLUSION

DeDoS is a new approach to defending against asymmetric DoS attacks. In DeDoS, software is built as a set of functional units called Minimal Splittable Units (MSUs) that can be replicated independently when under attack. DeDoS allows for more flexible allocation of resources and can efficiently dedicate more resources to MSUs under attack. Our evaluation shows that DeDoS runs with modest overheads and constitutes an effective defense against several state-of-the-art DoS attacks.

10 ACKNOWLEDGMENTS

This material is based upon work supported in parts by NSF Grants CNS-1527401, CNS-1513679, CNS-1563873, CNS-1703936, CNS-1453392, CNS-1513734, CNS-1704189, CNS 1750158, and CNS-1801884, as well as by the the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-16-C-0056, No. HR001117C0047 and No. HR0011-16-C0061. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF.

REFERENCES

[1] 2011. SSL Renegotiation DoS. (2011). https://www.ietf.org/mail-archive/web/tls/current/msg07553.html.
[2] 2017. DeDOS demonstration at SIGCOMM 2017. https://www.youtube.com/watch?v=KX4EPnUzDqk. https://www.youtube.com/watch?v=KX4EPnUzDqk
[3] 2017. Regular expression Denial of Service - ReDoS. (2017). https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.
[4] 2017. Tsung. http://tsung.erlang-projects.org/. http://tsung.erlang-projects.org/
[5] 2018. AWS Lambda. https://aws.amazon.com/lambda. https://aws.amazon.com/lambda
[6] 2018. Azure functions. https://functions.azure.com. https://functions.azure.com
[7] 2018. Common Vulnerabilities and Exposures (see CVE-2015-8386). (2018). http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8386.
[8] 2018. DeDOS on GitHub. https://github.com/dedos-project/DeDOS. https://github.com/dedos-project/DeDOS
[9] 2018. Google Cloud Functions. https://cloud.google.com/functions.
[10] 2018. OpenWhisk. https://developer.ibm.com/openwhisk.
[11] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.
[12] Ang Chen, Akshay Sriraman, Tavish Vaidya, Yuankai Zhang, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. 2016. Dispersing Asymmetric DDoS Attacks with SplitStack. In *Proc. HotNets*.
[13] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Nik Sultana, Bowen Wang, Jingyu Qian, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, et al. 2017. A Demonstration of the DeDoS Platform for Defusing Asymmetric DDoS Attacks in Data Centers. In *Proceedings of the SIGCOMM Posters and Demos*. ACM.
[14] F5. 2018. SSL Acceleration. https://f5.com/glossary/ssl-acceleration.
[15] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. 2015. Bohatei: Flexible and Elastic DDoS Defense. In *Proc. USENIX Security*.
[16] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. 1997. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proc. SOSP*.
[17] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads.. In *NSDI*. 363–376.

[18] Sean Gallagher. 2016. Double-dip Internet-of-Things Botnet Attack Felt Across the Internet. https://arstechnica.com/security/2016/10/double-dip-internet-of-things-botnet-attack-felt-across-the-internet/.
[19] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill Multiserver Approach. In *Proc 9th ACM SIGOPS European Workshop*. 109–114.
[20] Gkbrk. 2018. SlowLoris attack tool. https://github.com/gkbrk/slowloris.
[21] Saikat Guha, Paul Francis, and Nina Taft. 2008. *ShutUp: End-to-End Containment of Unwanted Traffic.* Technical Report. Cornell University.
[22] Quan Jia, Huangxin Wang, Dan Fleck, Fei Li, Angelos Stavrou, and Walter Powell. 2014. Catch Me if You Can: A Cloud-Enabled DDoS Defense. In *Proc. DSN*.
[23] Cheng Jin, Haining Wang, and Kang G. Shin. 2003. Hop-count filtering: an effective defense against spoofed DDoS traffic. In *Proc. CCS*.
[24] Joyent Inc. and other Node contributors. [n. d.]. NodeJS HTTP Parser. https://github.com/nodejs/http-parser.
[25] Christine Kern. 2016. Increased Use Of Multi-Vector DDoS Attacks Targeting Companies. (2016). http://www.bsminfo.com/doc/increased-use-of-multi-vector-ddos-attacks-targeting-companies-0001.
[26] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297.
[27] Soom Bum Lee, Min Suk Kang, and Virgil D. Gligor. 2013. CoDef: Collaborative Defense Against Large-Scale Link-Flooding Attacks. In *Proc. CoNEXT*.
[28] Qi Liao, David A. Cieslak, Aaron D. Striegel, and Nitesh V. Chawla. 2008. Using selective, short-term memory to improve resilience against DDoS exhaustion attacks. *Security and Communication Networks* 1, 4 (2008), 287–299.
[29] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Comm. ACM* 52, 11 (Nov. 2009), 87–95.
[30] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. 2002. Controlling High Bandwidth Aggregates in the Network. In *Proc. CCR*.
[31] Sam Newman. 2015. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.".
[32] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *Proc. SIGCOMM*.
[33] John Pescatore. 2014. *DDoS Attacks Advancing and Enduring: A SANS Survey.* Technical Report. SANS Institute.
[34] picoTCP 2018. picoTCP. http://www.picotcp.com/.
[35] Christian Rossow. 2014. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Proc. NDSS*.
[36] Fabrice J. Ryba, Matthew Orlinski, Matthias Wählisch, Christian Rossow, and Thomas C. Schmidt. 2015. Amplification and DRDoS Attack Defense – A Survey and New Perspectives. *CoRR* abs/1505.07892 (2015). http://arxiv.org/abs/1505.07892
[37] David Senecal. 2013. Slow DoS on the Rise. (2013). https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html.
[38] Willy Tarreau. 2018. HA-Proxy load balancer. http://haproxy.com/.
[39] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Proc. SOSP*.
[40] Jianxin Yan, Stephen Early, and Ross Anderson. 2000. The XenoService – A Distributed Defeat for Distributed Denial of Service. In *Proc. ISW*.
[41] Saman Taghavi Zargar, James Joshi, and David Tipper. 2013. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys & Tutorials* 15, 4 (2013), 2046–2069.